DB2 Data Management Software

IBM

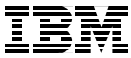
DB2 UDB's High Function Business Intelligence in e-business

- Exploit DB2's materialized views (ASTs/MQTs) feature
- Leverage DB2's statistics, analytic and OLAP functions
- Review sample business scenarios

Nagraj Alur
Peter Haas
Daniela Momiroska
Paul Read
Nicholas Summers
Virginia Totanes
Calisto Zuzarte

Redbooks

ibm.com/redbooks



International Technical Support Organization

DB2 UDB's High-Function Business Intelligence in e-business

September 2002

SG24-6546-00

Take Note! Before using this information and the product it supports, be sure to read the general information in “Notices” on page xvii.

First Edition (September 2002)

This edition applies to all operating system platforms that DB2 UDB Version 8 supports.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Examples

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®	DB2 OLAP Server™	Net.Data®
AS/400®	DB2 Universal Database™	Perform™
DataJoiner®	Everyplace™	QMF™
DataPropagator™	IBM®	Redbooks™
DB2®	IMS™	Redbooks(logo)™ 
DB2 Connect™	Intelligent Miner™	SP™
DB2 Extenders™	MVS™	WebSphere®

The following terms are trademarks of International Business Machines Corporation and Lotus Development Corporation in the United States, other countries, or both:

Lotus®	Approach®	K-station™
1-2-3®	Word Pro®	

The following terms are trademarks of other companies:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Preface

This IBM Redbook gives you an overview of the DB2 UDB engine's Business Intelligence (BI) functionality, and provides guidelines and examples of its use in real world business scenarios.

The focus is on DB2 UDB's materialized views feature (also known as Automatic Summary Tables [ASTs], or Materialized Query Tables [MQTs], as this feature is now called in DB2 product documentation), and its statistics, analytic, and OLAP functions.

Other BI functionality such as replication is not covered in this book.

This book is organized as follows:

1. **Chapter 1** provides an overview of IBM's Business Intelligence strategy and offerings, and positions the contents of this document with respect to the overall solution.
2. **Chapter 2** describes materialized views, and provides guidelines for creating, updating and tuning them.
3. **Chapter 3** provides an overview of DB2 UDB's statistics, analytic, and OLAP functions, with examples of their use.
4. **Chapter 4** describes typical business level queries that can be answered using DB2 UDB's statistics, analytic and OLAP functions. These business queries are categorized by industry, and describe the steps involved in resolving the query, with sample SQL and visualization of the results.
5. **Appendix A** is an introduction to statistics and analytic concepts used in resolving business queries described in Chapter 4.
6. **Appendix B** describes the DDL of the tables used in the examples.
7. **Appendix C** describes the main syntax elements of materialized view creation.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Nagraj Alur is a Project Leader with the IBM International Technical Support Organization, San Jose Center. He has more than 28 years of experience in DBMSs, and has been a programmer, systems analyst, project leader, consultant, and researcher. His areas of expertise include DBMSs, data warehousing, distributed systems management, and database performance, as well as client/server and Internet computing. He has written extensively on these subjects and has taught classes and presented at conferences all around the world. Before joining the ITSO in November 2001, he was on a two-year assignment from the Software Group to the IBM Almaden Research Center, where he worked on Data Links solutions and an eSourcing prototype.

Peter Haas received a Ph.D. in Operations Research from Stanford University in 1986. He has been a Research Staff Member at the IBM Almaden Research Center since 1987, where a major focus of his work has has been the application of probabilistic and statistical techniques to database systems. He invented a number of methods used by the DB2 query optimizer to estimate query processing costs, as well as methods for estimation of catalog statistics that have been incorporated into the DB2 RUNSTATS utility. He was a principal implementor of the correlation analyis and linear regression functions in DB2 UDB, and helped provided similar functionality to the DB2 Warehouse Manager product. He also developed technology for interactive online processing of complex aggregation queries, and led an effort to develop a prototype "online aggregation" interface to DB2. This work earned him the IBM Research Division 1999 Best Paper Award and an Honorable Mention at the 1999 ACM SIGMOD conference. Recently, he has been leading an effort to provide sampling and estimation functionality in DB2, and is helping to develop the proposed ISO standard for sampling in SQL queries. He is a member of the Institute for Operations Research and Management Sciences and is currently an Associate Editor for the journal "Operations Research". Since 1999, he has been affiliated with the Department of Management Science and Engineering at Stanford University, where he teaches a course on computer simulation.

Daniela Momiroska is an IT Specialist from Macedonia, working in Business Innovation Services in The Netherlands. She worked in IBM for three years as an HR Access developer, and in the last two years as a WebSphere Commerce Suite developer. Daniela has developed on-line shops for different customers, and has extensive knowledge of Internet based products centered around the WebSphere family. She holds a degree in computer science from the university of Skopje, Macedonia.

Paul Read is a Relational Database Specialist with 20 years of experience in application and system development and management. He is the lead EMEA technical professional for DB2 products on the INTEL platforms and Mobile platforms in the EMEA ATS PIC. He manages the beta and early support programs for DB2 UDB and Everyplace. He also provides technical consulting for the DB2 Brand software products across all platforms.

Nicolas Summers is a Senior IT Specialist working in Atlanta, Georgia for the IBM Americas Techline organization. His current responsibilities include technical sales support on DB2, Data Management, and Business Intelligence Solutions. He has over 17 years experience in the IT environment as a manufacturing engineer, systems engineer, sales specialist and technical sales support specialist. He holds degrees in Chemistry and Chemical Engineering.

Virginia Totanes is a DB2 UDB DBA IBM Certified Solutions Expert with over 20 years of experience in application development and support. Since joining IBM Canada's Data Warehousing practice (Business Intelligence) in 1999, she has assisted in building marketing data mart and CRM data warehouse. Virginia is knowledgeable of DB2 in MVS, AIX, Intel and AS/400 platforms.

Calisto Zuzarte is the manager of the DB2 Query Rewrite development group at the IBM Toronto Lab. His key interest is in the SQL Query performance area. His expertise spans the query rewrite and the cost based optimizer components, both key components within the SQL compiler, that impact query performance. Other development projects include features in DB2 involving data integrity constraints. He also manages the DB2 interests within the Centre for Advanced Studies (CAS) based in Toronto.

We would like to acknowledge the following people for their significant contributions to this project:

Vikas Krishna
Guy Lohman
Hamid Pirahesh
Richard Sidle
IBM Almaden Research Center

Michelle Jou
Bob Lyle
Swati Vora
IBM Silicon Valley Laboratory

Petrus Chan
Steve La
Bill O'Connell
IBM Toronto Laboratory

Aakash Bordia
Daniel DeKimpe
Gregor Meyer
IBM Silicon Valley Laboratory

Yvonne Lyon, for her technical editing and FrameMaker expertise
IBM International Technical Support Organization, San Jose Center

Notice

This publication is intended to help DB2 database administrators, application developers, and independent software vendors (ISVs) leverage DB2 UDB engine's powerful business intelligence functionality to achieve superior performance and scalability of their e-business applications. The information in this publication is not intended as the specification of any programming interfaces that are provided by DB2 UDB Version 8. See the PUBLICATIONS section of the IBM Programming Announcement for DB2 UDB Version 8 for more information about what publications are considered to be product documentation.

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to the address on page ii.

1

Business Intelligence overview

In this chapter we provide an overview of IBM's Business Intelligence strategy and offerings. We position the contents of this document with respect to the overall solution.

The topics covered include:

- e-business drivers
- IBM's BI strategy and offerings
- Redbook focus

1.1 e-business drivers

A number of factors have changed the business environment in recent years — from events such as global economics and competition, mergers and acquisitions, and savvy and demanding customers — to technology advances such as the World Wide Web, cheaper PCs and a host of Internet access devices such as PDAs, cellular telephones, and set tops. This is shown in Figure 1-1.

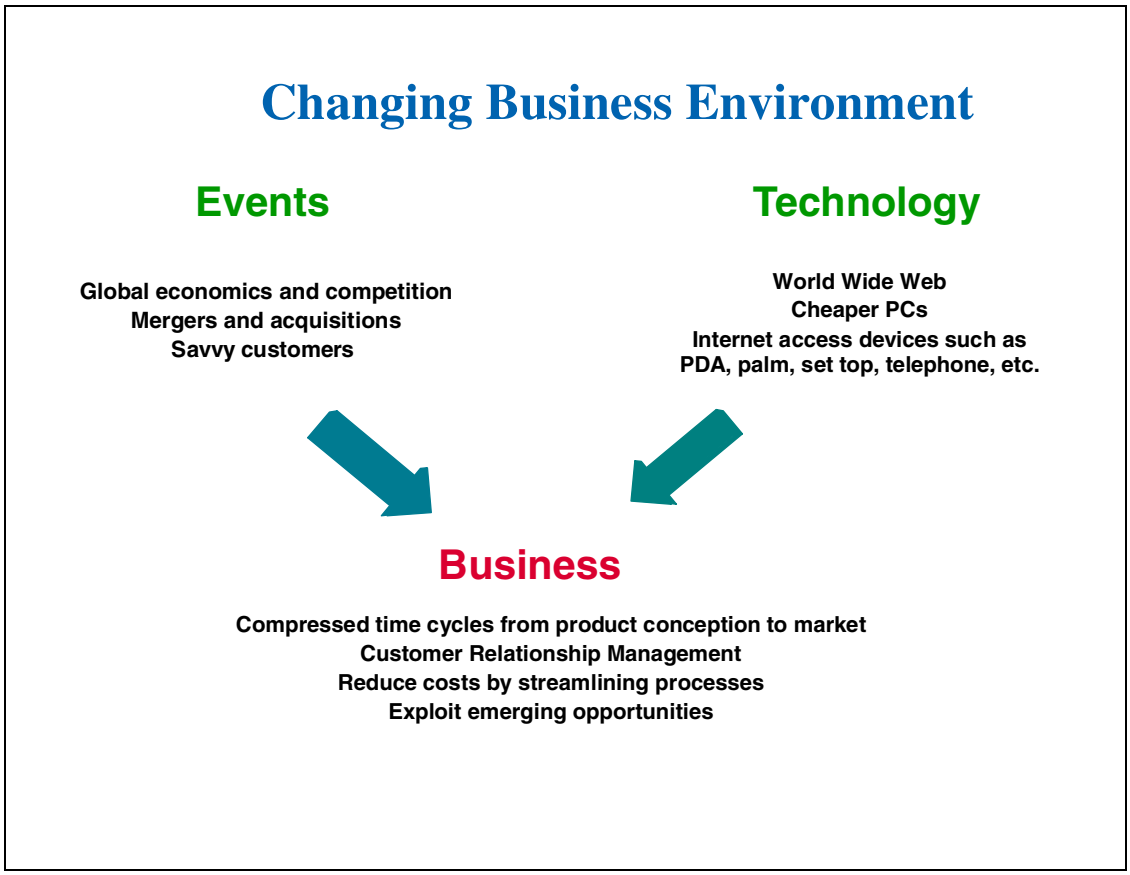


Figure 1-1 Changing business environment

Organizations clearly recognize that they can no longer dictate systems or clients, that the Internet cannot be controlled, and that downtime will impact more than employee productivity.

To survive and thrive in such an environment, organizations must adapt and innovate — business as usual could be a recipe for disaster. More so than ever before, the following issues are critical. It is now mandatory for a business to:

- Acquire and retain loyal and profitable customers. Businesses must become more responsive to its customers needs.

  Note: With government bodies, the emphasis is more on keeping its constituents happy — since acquisition and attrition are not relevant in most cases.

- Reduce costs by streamlining and transforming business processes, improving the productivity and efficiency of its employees and business partners and customers (self service and cutting out the intermediary — or disintermediation, as it is now called — is an important element here). While reducing costs is a perennial favorite that is generally characterized by stop and starts, this now takes on a new urgency.
- Become competitive with very short product conception to implementation to return on investment (ROI) cycles.
- Pursue every possible channel such as the Internet, and exploit emerging opportunities to ensure success and avoid failure.

Businesses recognize the urgency to transform various business processes for competitive advantage.

These processes are highlighted in Figure 1-2.

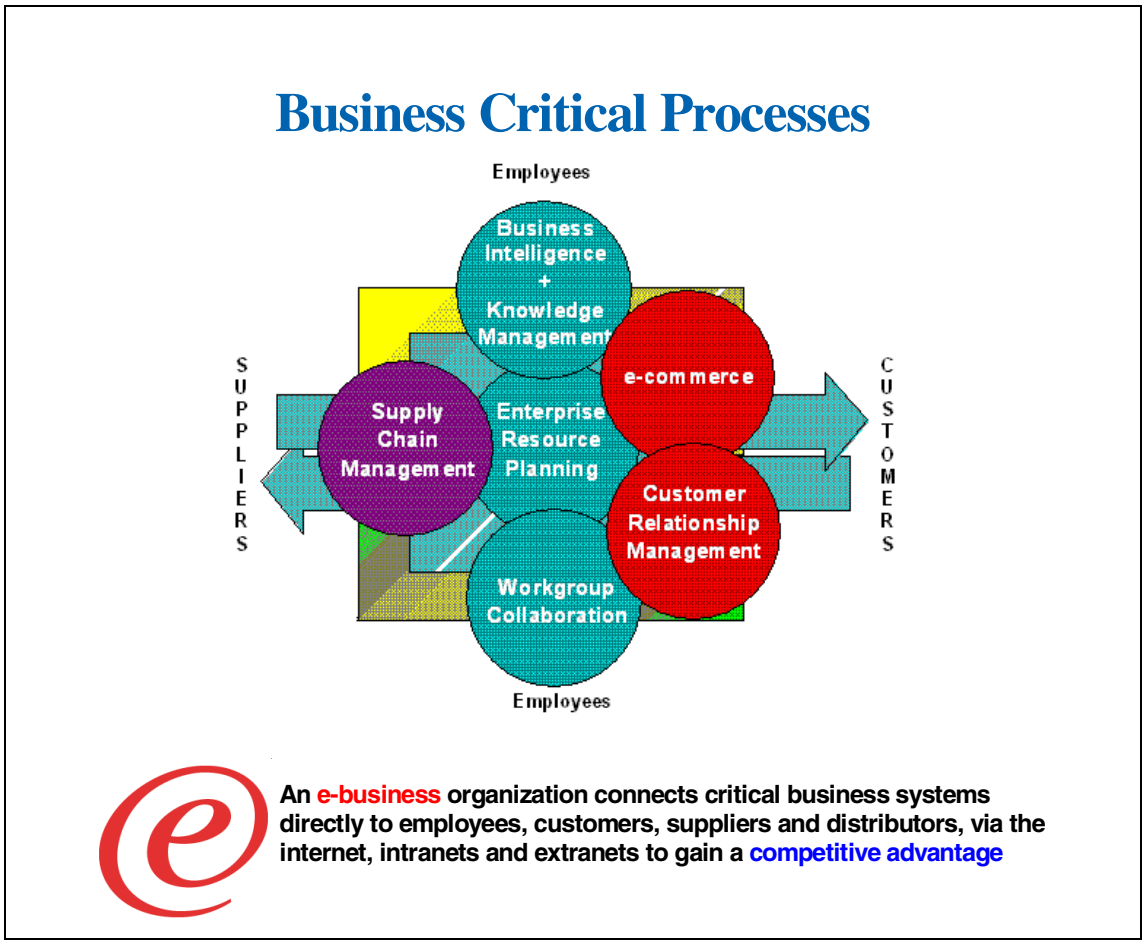


Figure 1-2 Business critical processes

These are the business critical processes we are concerned with:

- Customer Relationship Management (CRM), which has to do with identifying, understanding, anticipating and satisfying customer needs, that is, building loyalty through improved customer satisfaction.
- e-commerce, which is a new channel for an organization's goods and services to a whole wider global market.
- Supply Chain Management (SCM), which has to do with inter-company business processes. This involves improving the efficiency (and reducing costs) of interactions with suppliers, partners, distributors, customers, etc.

- Enterprise Resource Planning (ERP), which has to do with managing the bread-and-butter processes of an organization from planning, manufacturing, inventory, shipping/distribution, accounting and human resources.
- Workgroup collaboration, which has to do with sharing of resources and information amongst an organizations employees such as E-mail, meetings, document sharing, etc. Field Force Automation improves the productivity of employees in the field (salesman, technical support/maintenance persons, and delivery personnel) and improves customer satisfaction and responsiveness.
- **Business Intelligence (BI) which has to do with collecting and analyzing business information from a multitude of internal and external sources for competitive advantage.**
- Knowledge Management (KM), which has to do with combining and matching information and personnel skills to great effect.

Important: When an organization connects its business critical systems directly to customers, suppliers, distributors and employees in order to gain a competitive advantage, it transforms the organization and becomes an **e-business.**

By extending the reach of an organization's business critical systems using Internet technologies, the opportunity exists to gain significant competitive advantage by transforming:

- Employees — from competent to responsive individuals
- Customers — from one time interaction to lifetime loyalty through mass personalization instead of mass marketing
- Suppliers and distributors — from independence to interdependence

1.1.1 Impact of e-business

IT organizations have long understood their mission to support the applications required to meet business objectives via an infrastructure that delivers acceptable performance, availability, security, integrity and access to its user community. Most business organizations today have successfully implemented such infrastructures on private networks for their user community consisting of primarily their worldwide employees.

However, e-business results in the addition of customers, suppliers and distributors to the user community mix over the Internet, as well as intranets and extranets, accessing business critical systems. This is because the user community is now potentially global (requiring 7x24 operation), multi-lingual,

generally use a browser interface on a multiplicity of client platforms, have a lot of concerns over security and privacy, and can generate unpredictable workloads. Customers in particular have choices (they are just a mouse-click away from going over to the competition) and expect rapid response and superior service.

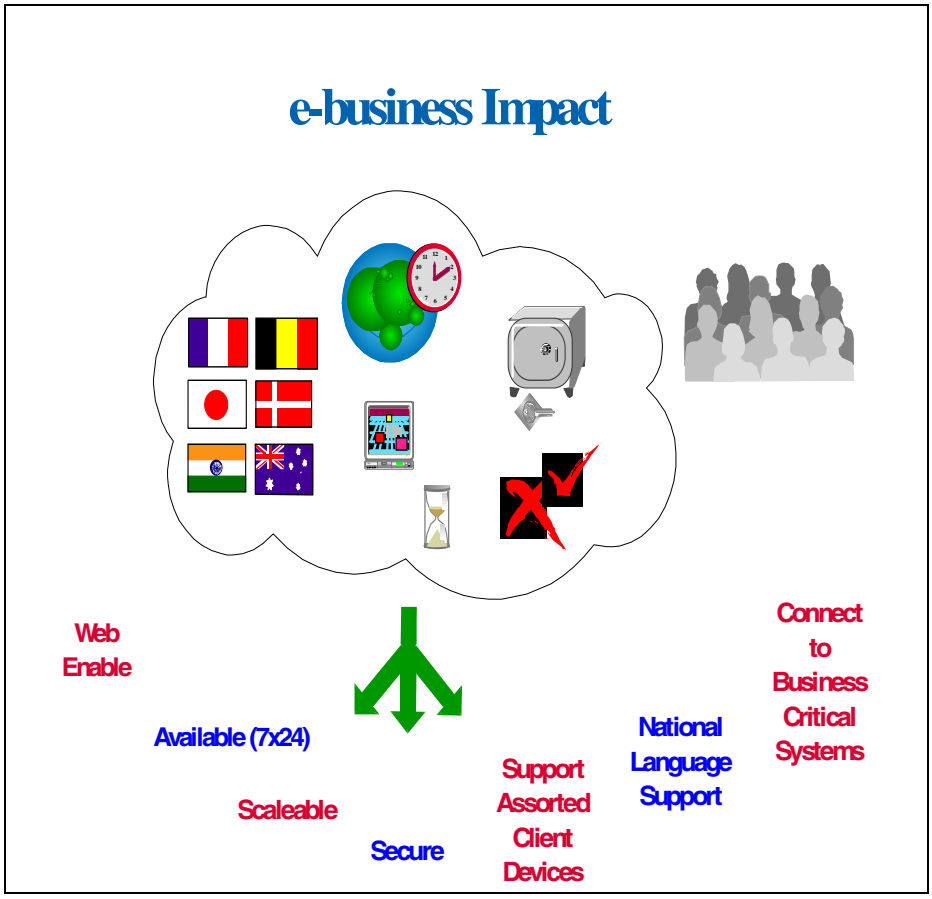


Figure 1-3 e-business impact

This has the potential to add orders of magnitude of complexity to the challenges of managing the demands of the exploding user community. e-business therefore exacerbates an already challenging situation.

Important: Of particular relevance to our upcoming discussion on BI is the unpredictability of the number and complexity of interactions from a user community that can test the most powerful servers (hardware and database) available.

Scalability is a critical requirement — both the ability for existing servers (hardware and database) to cope with spikes and workload growth, as well as the ability to painlessly augment existing server resources with more powerful servers or clusters of servers.

1.1.2 Importance of BI

In today's tough and competitive business climate, managers know that no matter what their core business, they are in the "information business". They must drive decisions that directly influence results, and realize that businesses that effectively use information to manage and impact decision making will have the greatest competitive edge.

Powerful transaction-oriented information systems are now commonplace in every major industry, effectively leveling the playing field for corporations around the world.

Industry leadership now requires analytically oriented systems that can revolutionize a company's ability to rediscover and utilize information they already own. These analytical systems derive insight from the wealth of data available, delivering information that's conclusive, fact-based and actionable. i.e. Business Intelligence.

Business intelligence can improve corporate performance in any information-intensive industry. Companies can enhance customer and supplier relationships, improve the profitability of products and services, create worthwhile new offerings, better manage risk, and pare expenses dramatically, among many other gains. Through business intelligence applications such as target marketing, customer profiling, and product or service usage analysis, businesses can finally begin using customer information as a competitive asset.

Having the right intelligence means having definitive answers to such key questions as these:

- Which of our customers are most profitable, and how can we expand relationships with them?
- Which of our customers provide us minimal profit, or cost us money?
- Which products and services can be cross-sold most effectively, and to whom?

- Which marketing campaigns have been most successful?
- Which sales channels are most effective for which products?
- How can we improve the caliber of our customers' overall experience?

Most companies have the raw data to answer these questions, since operational systems generate vast quantities of product, customer and market data from point-of-sale, reservations, customer service, and technical support systems. The challenge is to extract this information and reap its full potential.

Many companies take advantage of only a small fraction of their data for strategic analysis. The remaining untapped data, often combined with data from external sources like government reports, trade associations, analysts, the Internet, and purchased information, is a gold mine waiting to be explored, refined, and shaped into vital corporate knowledge. This knowledge can be applied in a number of ways, ranging from charting overall corporate strategy to communicating personally with employees, vendors, suppliers, and customers through call centers, kiosks, billing statements, the Internet, and other touch points that facilitate genuine one-to-one marketing on an unprecedented scale.

e-business impact on BI

Early business information systems were viewed as being standalone strategic decision making applications separate from operational systems that manage day to day business operations and supply data to the data warehouse and data marts.

However, the information and analyses provided by these systems have become vital to tactical day-to-day decision making as well, and are being integrated into the overall business processes.

In particular, e-business is:

- Encouraging this integration since organizations need to be able to react faster to changing business conditions in the e-business world than they do in the brick and mortar environment. The integration of business information systems into the overall business process can be achieved by building a closed loop decision making system in which the output of BI applications is routed to operational systems in the form of recommended actions such as product pricing changes for addressing specific business issues.
- Causing organizations to consider extending this closed loop process to the real-time automatic adjustment of business operations and marketing campaigns based on the decisions made by the BI system. Such a real-time closed loop system would deliver on-demand analytics for decision making capable of providing a significant competitive edge.

- Placing significant scalability requirements on a BI system because of e-business' characteristics of unpredictable workloads and voluminous data.

In the following section we provide an overview of IBM's BI strategy and product offerings, and focuses on key BI scalability characteristics in the DB2 Universal Database server product.

1.2 IBM's BI strategy and offerings

IBM has been a leader in BI, beginning with the landmark paper:

"An Architecture for a Business and Information System",
by Paul Murphy and Barry Devlin, 1988 IBM System Journal

Since then, IBM has invested significant development and marketing resources into the delivery of a set of technologies, products and partnerships for an integrated end-to-end enterprise analytics[1] solution.

IBM's enterprise analytics strategy has four main objectives:

- Support on-demand enterprise analytics and real-time decision making.
- Integrate BI and enterprise analytical processing into DB2 UDB.
- Simplify the building of an integrated system for delivering enterprise analytics.
- Form partnerships with leading BI and enterprise analytic vendors such as Ascential Software, Brio Technology, Business Objects, Evoke Software, Evolutionary Technology International (ETI), Hyperion Solutions, Trillium Software and Vality Technology.

Figure 1-4 describes the framework of an intelligent e-business and lists IBM's BI and enterprise analytics products. Also listed therein are the database requirements for BI which we will elaborate on in a following section.

[1] Enterprise Analytics is the collection of all critical business information metrics derived from a BI system, that is needed by executives, managers and business analysts to react rapidly to trends and changes in the marketplace

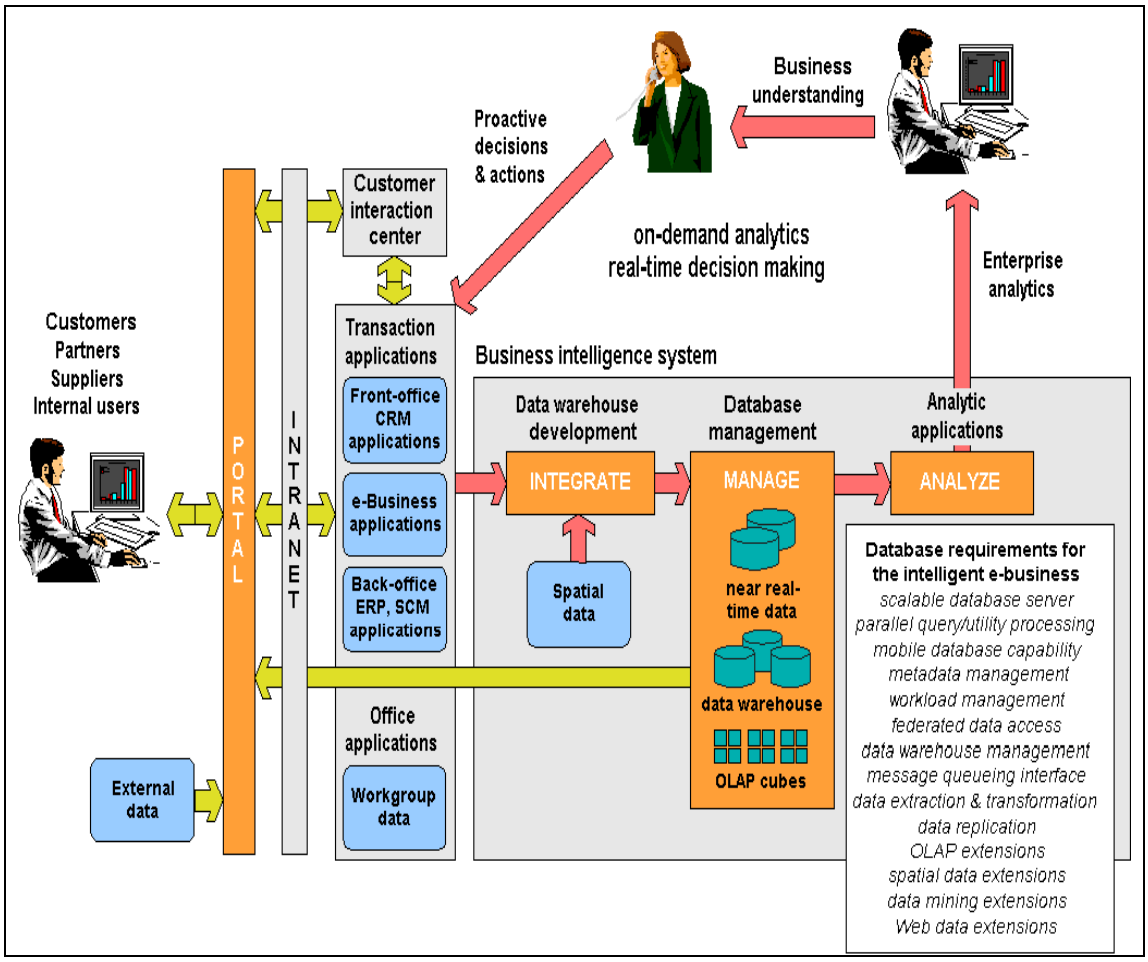


Figure 1-4 Intelligent e-business DataBase Associates International copyright

With reference to the above framework, IBM's products are as follows:

- Portal:
  - Federated data access products include DB2 Connect, DB2 Life Sciences Data Connect, DB2 Relational Connect, DB2 DataJoiner, DataJoiner Classic Connect, Data Links Manager, Net.Data, Data Warehouse development, DB2 Data Warehouse Manager, DB2 DataPropagator, IMS DataPropagator, Data Refresher and WebSphere MQ.
  - Portal development products include WebSphere Portal Server, Enterprise Information Portal and Lotus K-station and Discovery Server.

- Database Management products:
  - DB2 Universal Database integrated with DB2 Control Center, Data Warehouse Center, DB2 OLAP Starter Kit, DB2 Connect, DB2 Extenders for multimedia data, DB2 XML Extender, DB2 DataPropagator and Net.Data.
  - DB2 Everyplace, DB2 Query Patroller, DB2 Performance Monitor and DB2 Net Search Extender.
- Analytics:
  - Analytic applications such as WebSphere Commerce Suite, WebSphere Commerce Analyzer and WebSphere Site Analyzer.
  - Analytic application development products such as DB2 Intelligent Miner Scoring, DB2 Intelligent Miner for Data, DB2 Intelligent Miner for Text, DB2 OLAP Server, DB2 OLAP Server Analyzer, DB2 Spatial Extender, Query Management Facility and Visual Age for Java.

Please refer to the appropriate IBM product documentation for details about these products.

1.2.1 BI and analytic enhancements in DB2 UDB

A number of features have been added to DB2 UDB over the years to enhance the performance of BI and analytics, and thereby the scalability of the DB2 UDB engine particularly in an e-business environment of voluminous data, unpredictable workloads and rapidly changing business environments.

DB2 UDB BI and analytic performance features include SQL language extensions such as VARIANCE, COVARIANCE, CORRELATION, regression, ROLLUP and CUBE operators, support for large tables and bufferpools, partitioning of large tables for parallel processing, inter-query and intra-query parallel processing, materialized views for creating and refreshing summarized data, cost-based optimizer with multiple join algorithms, star schema optimization, index only access, index ANDing and ORing, caching of optimized SQL statements for repeat queries, and query rewrite of inefficient queries.

1.2.2 Advantages of BI functionality in the database engine

The advantages of executing BI functions in the database engine include:

- Superior scalability:

  By executing query requests as close to the data as possible, significant performance can be achieved thus enabling a larger workload to be serviced with the same amount of computing resources.

  Automatic exploitation of the database engine's parallelism capabilities adds significantly to performance and scalability of BI queries.

- Consistent results:

  Evaluating BI functions in the database engine guarantees consistent results and precision handling across all queries

- Reduced data latency:

  When BI functionality is executed by analytic tools, they typically operate on data that has been extracted and transformed from the operational system. There is a degree of latency introduced as a consequence. Also this intermediate extract/transform process may inhibit organizations from performing more frequent analytics.

  By supporting BI functions in the engine, organizations can potentially run analytics:

  – Directly against operational data thereby reducing the impact of latency.

  – More frequently against the data which could enable an organization to detect changing business conditions in a more timely fashion, thus gaining a competitive advantage.

- Reliability, availability, security and maintainability:

  A database engine's inherent manageability characteristics are available to all BI queries.

1.3 Redbook focus

In this redbook, we cover DB2 UDB engine features that have not been adequately discussed in either the DB2 technical library or other redbooks, but are critical to the performance and scalability of a BI system.

Our focus is on materialized views (aka Automatic Summary Tables [AST] or Materialized Query Tables [MQT] in IBM product documentation), and the statistics, analytic and OLAP functions in the DB2 UDB engine that can be exploited by analytic applications, DSS, OLAP and mining tools shown in Figure 1-5, to achieve superior performance and scalability.

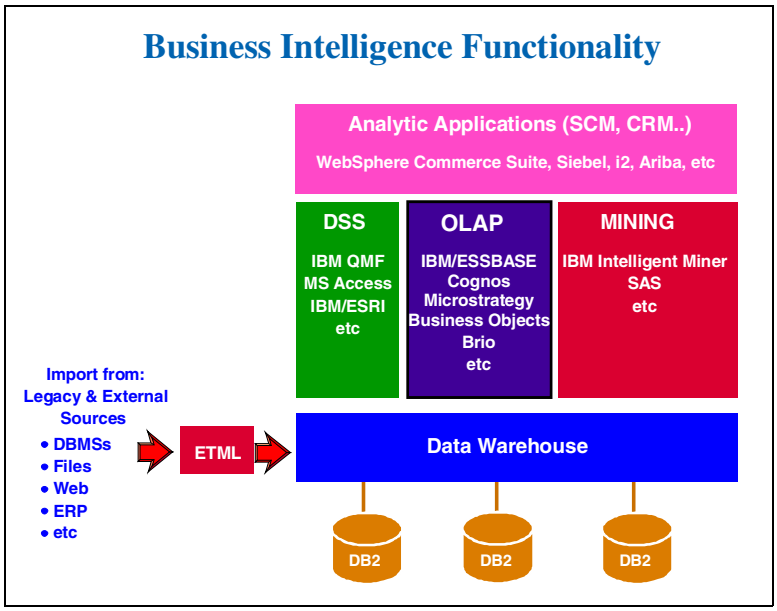


Figure 1-5 Business Intelligence functionality

1.3.1 Materialized views

Materialized views provide a powerful look aside capability for the DB2 optimizer to significantly improve the performance of complex and long running decision support queries over voluminous data. In some cases, the improvement has been in orders of magnitude!

Attention: Materialized views are known as Automatic Summary Tables (AST), or Materialized Query Tables (MQT) in IBM product documentation. We will use the more widely known term *materialized views* to mean ASTs/MQTs throughout this document.

The obvious challenges for the DBA are:

- Identifying the requirements for such materialized views
- Creating and maintaining them efficiently
- Dispensing with them when they are no longer necessary

These materialized view issues are covered in this book.

1.3.2 Statistics, analytic and OLAP functions

A number of SQL extensions have been implemented in DB2 UDB in support of key statistics, analytic and OLAP functions as described earlier, and this list is expected to grow over time.

Application developers and tool vendors who take advantage of DB2 UDB's SQL extensions can be expected to achieve significant performance benefits thus obtaining greater throughput and scalability, and a competitive edge.

In this book we discuss the various statistics, analytic, and OLAP functions supported in DB2 UDB, and provide examples of their usage in real world business scenarios.

Important: The actual performance benefits of using this functionality will depend upon an organization's unique workload and the resources available.

2

DB2 UDB's materialized views

In this chapter we provide an overview of DB2's materialized views, and offer guidelines for creating, updating, and tuning them.

The topics covered include:

- Materialized view overview
- Materialized view CREATE considerations
- Materialized view maintenance considerations
- Loading base tables (LOAD utility)
- Materialized view ALTER considerations
- Materialized view DROP considerations
- Materialized view matching considerations
- Materialized view design considerations
- Materialized view tuning considerations
- Refresh optimization
- Materialized view limitations
- Replicated tables in nodegroups

2.1 Materialized view overview

In this section, we describe:

- Materialized view motivation
- Materialized view concept overview
- Materialized view usage considerations
- Materialized view terminology

2.1.1 Materialized view motivation

Before we explain what a materialized view is, let us discuss the motivation for its introduction.

In a data warehouse environment, users often issue queries repetitively against large volumes of data with minor variations in a query's predicates. For example:

- Query A might request the number of items belonging to a consumer electronics product group sold in each month of the previous year for the western region.
- Query B may request the same kind of information for only the month of December for all regions in the USA.
- Query C might request monthly information for laptops for all regions in the USA over the past 6 months.

The results of these queries are almost always expressed as summaries or aggregates. The base data could easily involve millions of transactions stored in one or more tables, that would need to be scanned repeatedly to answer these queries. Query performance is more than likely to be poor in such cases.

Materialized views were introduced to address the aforementioned performance problems.

2.1.2 Materialized view concept overview

Figure 2-1 provides an overview of the materialized view concept.

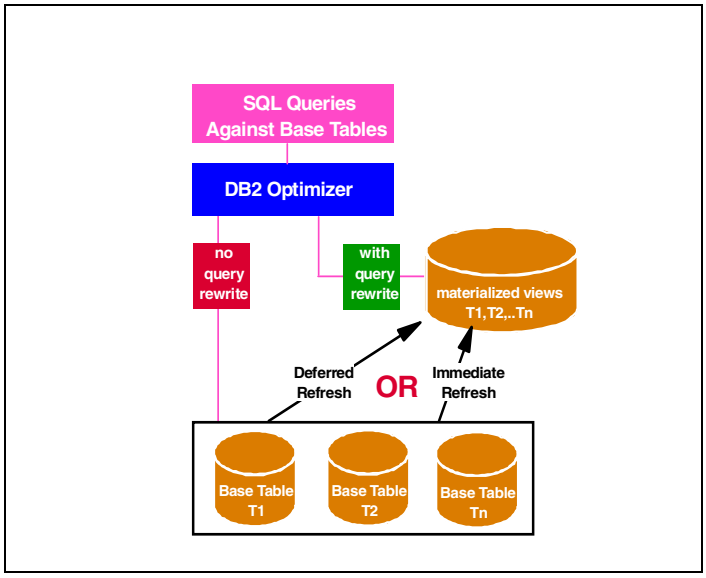


Figure 2-1 Materialized view overview

Support for materialized views requires the following:

- Having a DBA **pre-compute** an aggregate query, and **materialize** the results into a table. This summary table would contain a superset of the information that could answer a number of queries that had minor variations.
- Enhancing the DB2 optimizer to **automatically rewrite** a query against the base tables, to target the materialized view instead (if appropriate) in order to satisfy the original query.

Since the materialized view often contains precomputed summaries and/or a filtered subset of the data, it would tend to be much smaller in size than the base tables from which it was derived. When a user query accessing the base table is automatically rewritten by the DB2 optimizer to access the materialized view instead, then significant performance gains can be achieved.

Case Study: In one customer scenario, a query required computing the total sales for all product categories for the year 1998. It involved joining a 1.5 billion row transaction table with three dimensional tables. The query had to touch at least 400 million rows in the transaction table.

Without an materialized view, the response time on a 22-node SP was 43 minutes.

With an materialized view, the response time was reduced to 3 seconds!!!

In this case, DB2 touched at least 4000 times fewer rows and avoided a join.

The benefits achievable with materialized views depends upon one's own unique workload, and your mileage will vary.

Important: Materialized view functionality is somewhat similar to the role of a DB2 index which provides an efficient access path that the query user is typically unaware of. However, unlike an index, a user may directly query the materialized view, but this is not generally recommended since it would detract from the appeal of an materialized view being a black box that an administrator creates and destroys as required to deliver superior query performance.

Adapting their queries to use a materialized view may not be a trivial exercise for the user.

Figure 2-1 also shows that two approaches may be adopted to maintain the materialized view:

- In the deferred refresh approach, DB2 does not automatically keep the materialized view in sync with the base tables, when the base tables are updated. In such cases, there may be a latency between the contents of the materialized view, and the contents in the base tables.
- In the immediate refresh approach, the contents of the materialized view are always kept in sync with the base tables. An update to an underlying base table is immediately reflected in the materialized view as part of the update processing. Other users can see these changes after the unit of work has completed on a commit. There is no latency between the contents of the materialized view and the contents in the base tables.

The pros and cons of each approach are discussed in detail in 2.3, "Materialized view maintenance considerations" on page 26.

Example 2-1 and Example 2-2 show examples of creating a materialized view. Details of the syntax are described in Appendix C-1, “Main syntax elements of materialized views” on page 242.

Example 2-1 Example of creating a deferred refresh materialized view

```
CREATE TABLE custtrans AS
(
SELECT cust_id, COUNT(*) AS counttrans
FROM trans
WHERE cust_id > 500
GROUP BY cust_id
)
DATA INTITIALLY DEFERRED REFRESH DEFERRED
```

Example 2-2 Example of creating a refresh immediate materialized view

```
CREATE TABLE dba.trans_agg AS
(
SELECT ti.pgid, t.locid, t.acctid, t.status,
             YEAR(pdate) as year, MONTH(pdate) AS month,
             SUM(ti.amount) AS amount, COUNT(*) AS count
FROM transitem AS ti, trans AS t
WHERE  ti.transid = t.transid
GROUP BY YEAR(pdate), MONTH(pdate)
)
DATA INITIALLY DEFERRED REFRESH IMMEDIATE
```

Note: In both these examples, it is the “DATA INITIALLY DEFERRED REFRESH DEFERRED | IMMEDIATE” that identifies it as a materialized view. The SUMMARY keyword that was part of the earlier syntax for creating materialized views is not required, but is supported for backward compatibility.

A number of other parameters such as DISABLE | ENABLE QUERY OPTIMIZATION, MAINTAINED BY SYSTEM | USER, PROPAGATE IMMEDIATE etc. are applicable to materialized views, and are allowed to default in the examples shown.

2.1.3 Materialized view usage considerations

The decision to implement materialized views will depend upon answers to the following questions:

1. Is it acceptable for the user application if the user query gets different results depending on whether the query routes to the materialized view, or accesses the base tables directly?

2. What is the acceptable latency of data for the query?
   - For data warehouses and strategic decision making, there can be (and in some cases needs to be) a certain latency such as end-of-day or end-of-week or end-of-month information. In such cases, the materialized views need not be kept in sync with the base tables. DB2 supports a deferred refresh of the materialized view for such scenarios
   - For OLTP and tactical decision making, any materialized view latency may be unacceptable, and DB2 supports an immediate refresh of the materialized view in such cases. It is important to note that there could be significant performance overheads for updates of the base tables when the volume of update activity is high in these scenarios.
3. Will the performance benefits of implementing materialized views be significant enough to offset the overheads of maintaining them. Overheads include disk space for the staging table[1] (if one exists), materialized view and any associated indexes, and cost of maintaining the materialized view when the underlying base tables are updated.

2.1.4 Materialized view terminology

While materialized views involving aggregate tables were initially introduced to address performance problems in a data warehouse environment, they can also be exploited for better performance in an e-business environment. For example, in e-commerce, product catalog information can be cached on mid-tier servers to significantly improve the performance of catalog browsing. Materialized views can be used to cache back-end database product information on a mid-tier, and such materialized views do not involve summary data. DB2 supports such caching by allowing materialized views to be defined over nicknames that are used to define a remote table. Populating the materialized view involves pulling data from the remote table and storing it locally, resulting in significant performance benefits. Note that this only applies to deferred refresh materialized views.

[1] See “Incremental refresh” on page 29 for further details.

Materialized views were introduced in DB2 UDB Version 5, and have been continuously enhanced with new capabilities since then.

Note: Prior to DB2 UDB V8, materialized views were named Automatic Summary Tables (ASTs) in IBM product documentation. While it was possible to define non-aggregate materialized views in DB2 UDB V7, the restriction was that such a materialized view could only be defined on single base table. In DB2 UDB V8, this restriction has been removed, and since materialized views can include other than summary data, the more generalized term Materialized Query Tables (MQT) was introduced. ASTs can be considered to be a subset of the generalized MQT which specifically includes summary data.

Attention: Throughout this document, we use the more widely known term *materialized views* to mean ASTs/MQTs.

2.2 Materialized view CREATE considerations

Figure 2-2 provides an overview of the steps involved in creating a materialized view.

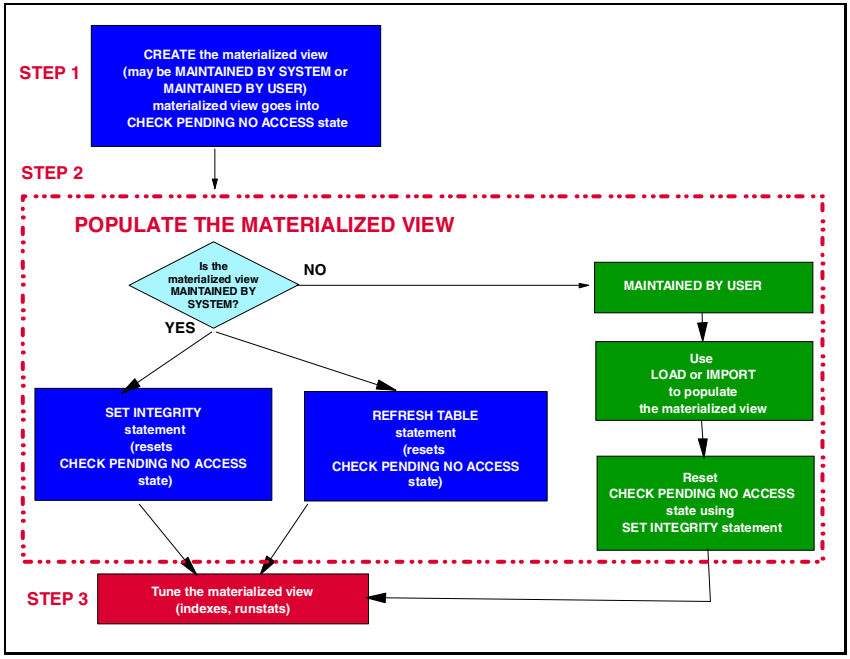


Figure 2-2 CREATE materialized view overview

We now briefly describe these steps:

2.2.1 Step 1: Create the materialized view

Assuming that the user has determined what the materialized view should look like (see 2.8, “Materialized view design considerations” on page 60 for a discussion of the design considerations), the following occurs when the materialized view creation DDL is executed.

- Since the materialized view has not yet been populated, it is placed in CHECK PENDING NO ACCESS[2] state regardless of whether it is a system maintained or a user maintained materialized view. No SQL read or write access is permitted against tables in a CHECK PENDING NO ACCESS state.
- Dependencies regarding the base tables and the materialized view are recorded in SYSCAT.TABLES, SYSCAT.TABDEP, SYSCAT.VIEWS just as any other table or view definition creation.
- All packages that update the base tables on which the materialized view is built are invalidated if the REFRESH IMMEDIATE option is chosen, or it is a REFRESH DEFERRED materialized view that is associated with a staging table. This is because the SQL compiler must add appropriate operations in the package to support the refresh immediate materialized views or staging tables. When the package is first accessed after invalidation, an automatic rebind ensures that the package has been updated to support the materialized view or staging table.

  An EXPLAIN of the rebound package will highlight the additional SQL operations being performed to support materialized views.

2.2.2 Step 2: Populate the materialized view

DB2 supports materialized views that are either:

- **MAINTAINED BY SYSTEM (default):** In this case, DB2 ensures that the materialized views are updated, when the base tables on which they are created get updated. Such materialized views may be defined as either REFRESH IMMEDIATE, or REFRESH DEFERRED. If the REFRESH DEFERRED option is chosen, then either the INCREMENTAL or NON INCREMENTAL refresh option can be chosen during refresh. This is discussed in detail in 2.3, “Materialized view maintenance considerations” on page 26.
- **MAINTAINED BY USER:** In this case, it is up to the user to maintain the materialized view whenever changes occur to the base tables. Such materialized views *must be* defined with the REFRESH DEFERRED option.

[2] This was previously called the CHECK PENDING state.

Even though the REFRESH DEFERRED option is required, unlike MAINTAINED BY SYSTEM, the INCREMENTAL or NON INCREMENTAL option does *not* apply to such materialized views, since DB2 does not maintain such materialized views.

A couple of possible scenarios where such materialized views could be defined are as follows:

a. For efficiency reasons, when the user is convinced that (s)he can implement materialized view maintenance far more efficiently than the mechanisms used by DB2. For example, the user has high performance tools for rapid extraction of data from base tables, and loading the extracted data into the materialized view.

b. For leveraging existing "user maintained" summaries, where the user wants DB2 to automatically consider them for optimization for new ad hoc queries being executed against the base tables.

Appendix C, "Materialized view syntax elements" on page 241 provides details about the syntax, etc.

Populating a MAINTAINED BY SYSTEM materialized view

Typically, one of the following two approaches can be used to populate a MAINTAINED BY SYSTEM materialized view. These are described briefly as follows:

1. SET INTEGRITY

   The following statement causes the materialized view to be populated, and results in the CHECK PENDING NO ACCESS state being reset on successful completion.

   ```
   SET INTEGRITY FOR tablename IMMEDIATE CHECKED[3]
   ```

   Note: SET INTEGRITY statement also applies to staging tables (see 2.3.1, "Deferred refresh" on page 27 for details on staging tables).

2. REFRESH TABLE

   The following statement also causes the materialized view to be populated, and the CHECK PENDING NO ACCESS state to be reset on successful completion.

   ```
   REFRESH TABLE tablename
   ```

[3] Since we do not have constraints on materialized views, we should not specify exception tables for materialized views.

Note: There is no semantic difference between using the SET INTEGRITY or the REFRESH TABLE syntax; both are treated identically.

Attention: Using the SET INTEGRITY or REFRESH TABLE approaches to populate the materialized view involves using SQL INSERT subselect type processing, which may result in excessive logging when very large materialized views are being populated.

Users may want to avoid this logging overhead during the *initial population* of the materialized view by following these steps:

1. Make the base tables read only.
2. Extract the required data from the base tables, and write it to an external file.
3. IMPORT or LOAD the extracted data into the materialized view. These operations are permitted on a table in CHECK PENDING NO ACCESS state.
4. Reset the CHECK PENDING NO ACCESS state on the materialized view using the following statement:

   ```
   SET INTEGRITY FOR tablename ALL IMMEDIATE UNCHECKED
   ```

5. Make the base tables read/write.

Important: With this approach, it is the user's responsibility to ensure that the data being loaded into the materialized view correctly matches the query definition of the materialized view.

Another option, which overcomes the concern of extracting data correctly to an external file, is to use the LOAD the data from a cursor, as shown in Example 2-3.

Example 2-3 LOADing from a cursor

```
-- base table definition
CREATE TABLE t1
   (
   c1 INT,
   c2 INT,
   c3 INT
   );

-- create the refresh deferred materialized view
CREATE TABLE a1 AS
   (
   SELECT c1, count(*) AS cs
```

```
   FROM t1
   GROUP BY c1
   )
DATA INITIALLY DEFERRED REFRESH DEFERRED;

-- make the base table read only

-- ensure that the cursor is not closed automatically
UPDATE COMMAND OPTIONS USING C OFF;

-- cursor declaration that mirrors materialized view query definition
DECLARE cc CURSOR FOR
   SELECT c1, COUNT(*)
   FROM t1
   GROUP BY c1;

-- LOAD the materialized view with the contents of the cursor
LOAD FROM CC OF CURSOR REPLACE INTO a1;

-- reset the CHECK PENDING NO ACCESS state onthe materialized view
SET INTEGRITY FOR a1 ALL IMMEDIATE UNCHECKED;

-- make the base table read/wite
```

When incremental refresh[4] is supported for such materialized views, REFRESH TABLE can subsequently be used to perform the necessary ongoing maintenance of this materialized view when the underlying tables are updated.

Note: When incremental refresh is not supported for such materialized views, it may be more appropriate to create materialized views as MAINTAINED BY USER, and adopt the following approach for populating user maintained materialized views.

Populating a MAINTAINED BY USER materialized view

In the user-managed approach, it is the user's responsibility to populate the materialized view, and make it available for matching by resetting the CHECK PENDING NO ACCESS state. The user is responsible for ensuring the consistency and integrity of the materialized view. Typically, the user would:

1. Make the base tables read only.
2. Extract the required data from the base tables, and write it to an external file.

[4] See 2.3.1, "Deferred refresh" on page 27 for a discussion of incremental refresh.

3. IMPORT or LOAD the data into the materialized view — these operations are permitted on a table in CHECK PENDING NO ACCESS state.

Important: SQL INSERT statements cannot be issued against a table in CHECK PENDING NO ACCESS state. If the user wishes to populate the materialized view using SQL INSERT statements, then (s)he must first reset the CHECK PENDING NO ACCESS state. However, optimization must first be disabled via the DISABLE QUERY OPTIMIZATION option in the CREATE DDL to ensure that a dynamic SQL query does not accidentally optimize to this materialized view while the data in it is still in a state of flux. Once the materialized view has been populated, then optimization needs to be enabled.

4. Reset the CHECK PENDING NO ACCESS state using the following statement:

```
SET INTEGRITY FOR tablename ALL IMMEDIATE UNCHECKED
```

   This action is recorded in the CONST_CHECKED column[5] (value 'U') in the catalog table SYSCAT.TABLES column, indicating that the user has assumed responsibility for data integrity of the materialized view.

5. Make the base tables read/write.

The process described in Example 2-3 can also be used to populate a maintained by user materialized view.

2.2.3 Step 3: Tune the materialized view

This involves the creation of appropriate indexes, and executing the RUNSTATS utility on the materialized view to ensure optimal access path selection. "Materialized view tuning considerations" on page 87 describes these considerations in greater detail.

2.3 Materialized view maintenance considerations

DB2 supports two approaches for maintaining materialized views — a deferred refresh approach, and an immediate refresh approach. These approaches are described in this section.

Table 2-1 summarizes some considerations relating to refresh immediate and refresh deferred materialized views.

[5] CONST_CHECKED is defined as a CHAR(32) and is viewed as an array, where the value stored at CONST_CHECKED(5) represents a summary table.

Table 2-1 Refresh considerations

Items	REFRESH IMMEDIATE	REFRESH DEFERRED	
	System maintained only	**System maintained**	**User maintained only**
Static SQL	Optimization	No optimization	No optimization
Dynamic SQL	Optimization	Optimization	Optimization
SQL INSERT, UPDATE, DELETE against materialized view	Not permitted	Not permitted	Permitted
REFRESH TABLE tablename	Permitted	Permitted	Not applicable
REFRESH TABLE NOT INCREMENTAL	Permitted	Permitted	Not applicable
REFRESH TABLE INCREMENTAL	Permitted	Requires staging table	Not applicable
Staging table	Not applicable	Same restrictions to creating them, as those applying to REFRESH IMMEDIATE materialized views	Not applicable

2.3.1 Deferred refresh

This maintenance approach is used when the materialized view need not be kept in sync with the base tables as the base tables are being updated. The data may be refreshed when appropriate as deemed by the administrator. Such materialized views are called REFRESH DEFERRED tables.

Example 2-4 shows an example of SQL for creating a REFRESH DEFERRED materialized view.

Example 2-4 Creating a refresh deferred materialized view

```
CREATE SUMMARY TABLE dba.summary_sales
   AS (SELECT ...........)
   DATA INITIALLY DEFERRED
   REFRESH DEFERRED
```

Restriction: Materialized view optimization does not occur for static SQL statements with REFRESH DEFERRED tables.

Figure 2-3 provides an overview of maintaining a REFRESH DEFERRED materialized view.

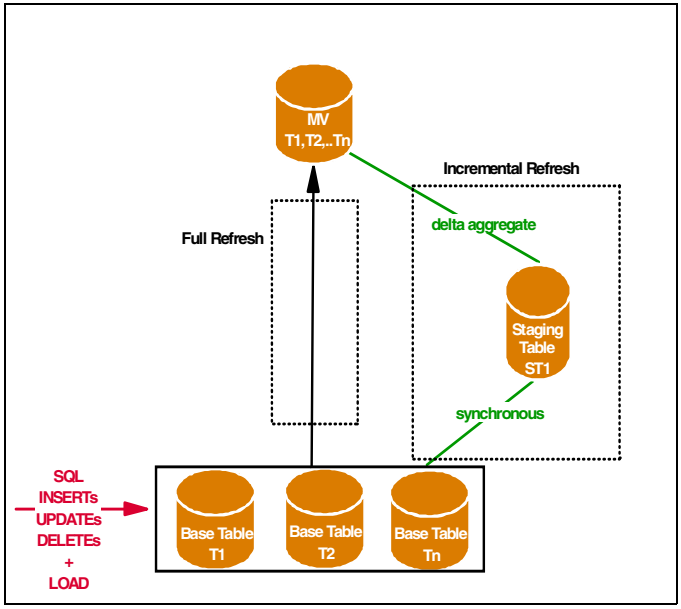


Figure 2-3 Deferred refresh

A REFRESH DEFERRED materialized view may be maintained via a REFRESH TABLE command with either a full refresh (NOT INCREMENTAL) option, or an incremental refresh (INCREMENTAL) option.

- With a full refresh, DB2 deletes the contents of the contents of the materialized view, scans the base table(s), computes and generates all the necessary rows, and then inserts these rows in to the materialized view.
- With an incremental refresh, a staging table[6] *must be* defined for the materialized view. DB2 synchronously updates the staging table as the base tables are being updated, and then computes the delta joins and aggregates for updating the materialized view when the incremental refresh is requested.

Important: Only MAINTAINED BY SYSTEM materialized views support the REFRESH TABLE command.

[6] See “Incremental refresh” on page 29 for further details.

A brief discussion of the full refresh and incremental refresh follows:

Full refresh

The following statement will request a full refresh of the materialized view dba.summary_sales:

```
REFRESH TABLE dba.summary_sales NOT INCREMENTAL
```

The NOT INCREMENTAL option specifies a full refresh for the materialized view by recomputing the materialized view definition. When this is done, all existing data within the table is deleted, and the query defining the materialized query table is computed in it's entirety. For the duration of a full refresh, DB2 takes a share lock on the base tables, and a z-lock on the materialized view. Depending upon the size of the base tables, this process can take a long time. The base tables are not updatable for this duration, and the materialized view is not available for access or optimization. An additional consideration is that significant logging may occur during a full refresh as it populates the materialized view.

Incremental refresh

The following statement will request an incremental refresh of the materialized view dba.summary_sales:

```
REFRESH TABLE dba.sales_summary INCREMENTAL
```

The INCREMENTAL option specifies an incremental refresh for the materialized view by considering only the consistent content of an associated staging table.

Note: If DB2 detects that the materialized view needs to be fully recomputed, then an error condition is returned.

Important: If neither INCREMENTAL nor NOT INCREMENTAL is specified on the REFRESH TABLE statement, the system will determine whether incremental processing is possible. If not possible, full refresh will be used. The following actions apply:

- If a staging table is present for the materialized view that is to be refreshed, and incremental processing is not possible because the staging table is in a pending state, an error is returned.
- Full refresh will be performed if the staging table is inconsistent and the staging table is pruned.
- Incremental refresh will be performed using the contents of a valid staging table, and the staging table will be pruned.

Figure 2-4 provides an overview of the steps involved in creating, enabling and exploiting a staging table on the materialized view shown in Example 2-5.

Example 2-5 Materialized view with REFRESH DEFERRED option

```
CREATE SUMMARY TABLE summary_sales
   AS (SELECT ..........)
   DATA INITIALLY DEFERRED
   REFRESH DEFERRED
```

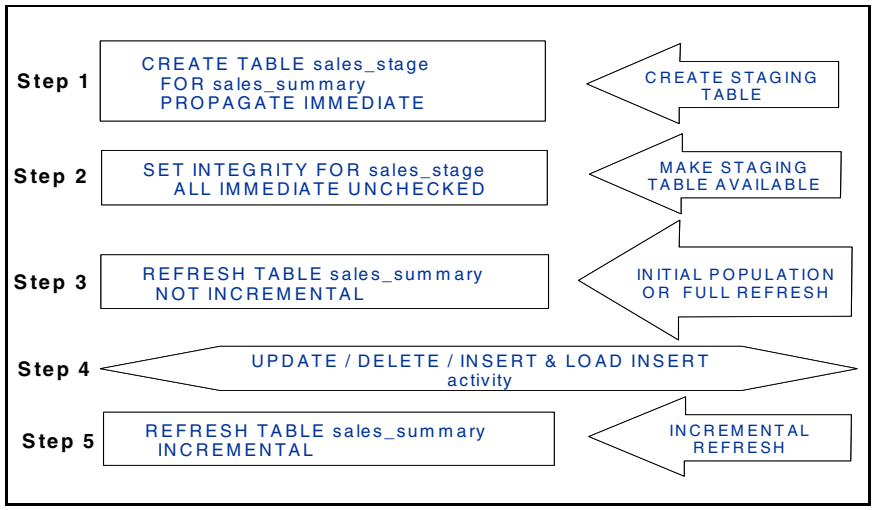


Figure 2-4 Incremental refresh with staging table

We discuss each of these steps briefly.

Step 1

The staging table is created on the previously defined materialized view in this step.

Restriction: For this to be successful, even though the *sales_summary* table is defined with the REFRESH DEFERRED option, it must satisfy all the conditions of a REFRESH IMMEDIATE materialized view as described in “Materialized view limitations” on page 92.

Note: Both the materialized view sales_summary and the staging table sales_stage are in a CHECK PENDING NO ACCESS state.

The PROPAGATE IMMEDIATE parameter in the CREATE TABLE statement indicates that any changes made to the base tables as part of an INSERT, DELETE, UPDATE operation are immediately added to the staging table, with additional information generated in the three extra columns of the staging table.This is done as part of the same SQL statement.

The schema of the staging table looks much like the materialized view for which it has been defined. The difference is that the staging table may have two or three more columns than its associated materialized view. These additional columns are as follows:

- GLOBALTRANSID CHAR(8) — global transaction ID for each propagated row
- GLOBALTRANSTIME CHAR(13) — the timestamp of the transaction
- OPERATIONTYPE INT — values -1, 0 and 1 for SQL DELETE, UPDATE and INSERT respectively

Restriction: Each column name in the staging table must be unique and unqualified. If a list of column names is not specified, the columns of the table inherit the names of the columns of the associated summary table and the additional columns are defined. If a list of columns is specified, it has to include the required extra columns.

For replicated[7] materialized views and non-aggregate query materialized views, the staging table contains three more columns than the associated materialized view. Otherwise, the staging table only contains two extra columns, with the OPERATIONTYPE column being omitted.

Step 2

Issuing the SET INTEGRITY statement against the staging table takes it out of the CHECK PENDING NO ACCESS state, thus making it available for processing.

Step 3

Issuing the REFRESH TABLE statement against the materialized view sales_summary with the NOT INCREMENTAL option, populates the materialized view and takes it out of CHECK PENDING NO ACCESS state on successful completion.

Step 4

This step reflects ongoing update activity against the underlying base tables. For SQL operations, the staging table is updated synchronously within the same unit of work as the SQL INSERT, UPDATE or DELETE statement.

[7] This applies to a partitioned database environment.

Note: When LOAD INSERT is used against the base table(s), the staging table is *not* updated synchronously unlike the case of SQL statements.

The following steps will synchronize the staging table with the contents of the base table(s).

1. SET INTEGRITY FOR base_table_name IMMEDIATE CHECKED

   This statement will make the base table available after verifying the consistency of the newly appended rows to the base table. It will also cause a CHECK PENDING NO ACCESS state to be set on the staging table. This is to ensure that a subsequent REFRESH TABLE on the materialized view with an INCREMENTAL option will fail, since the staging table is not synchronized with the base table.

2. SET INTEGRITY FOR staging_table_name IMMEDIATE CHECKED

   This statement causes the staging table to be synchronized with the delta changes of LOAD INSERT activity, and makes the staging table available by removing the CHECK PENDING NO ACCESS state.

Step 5

When the user issues the following REFRESH TABLE statement against the materialized view with the INCREMENTAL option, DB2 uses the data in the staging table if possible, to update the target materialized view and prune the processed data in the staging table as part of this process.

```
REFRESH TABLE dba.summary_sales INCREMENTAL
```

The rows in the staging table are grouped and consolidated as required, before the changes are applied to the materialized view.

Important: DB2 takes a z-lock on the materialized view and the staging table (if one exists) during the REFRESH TABLE statement. If the staging table is unavailable for extended periods of time because of a lengthy refresh, then it has the potential to negatively impact update activity on the base tables. Similarly, having the materialized view unavailable for an extended period because of refresh times can negatively affect the performance of queries accessing the materialized view.

Note that until the materialized view is refreshed via a REFRESH TABLE statement, the content of the staging table reflects the delta changes to the base table(s) since the last REFRESH TABLE statement.

Deferred refresh considerations

The frequency of execution of the REFRESH TABLE statement has an impact on the following:

- **Latency of the data:** The tolerance for this is dependent upon the application.
- **Logging overhead:** More frequent refreshes have the potential to involve more updates against the materialized view. Less frequent refreshes may result in fewer updates because data consolidation may occur either on the staging table or the base table.

  Logging space can be of concern when large volumes of data are involved in refreshing a materialized view. During refresh, rows have to be inserted/updated into the materialized view. If a staging table exists, it has to be pruned as well. This contributes to logging overhead. Either one of the following approaches may alleviate this problem.

  – The preferred approach would be to use the ALTER TABLE ..... NOT LOGGED INITIALLY option to avoid logging during the refresh. By limiting the unit-of-work to the REFRESH TABLE statement, the probability of an inadvertent rollback due to an error is quite small. In the unlikely case that a rollback does occur, the materialized view can be refreshed again. However, only a full refresh occurs in such cases (no incremental refresh is possible), and the database administrator will have to drop and recreate the materialized view DDL definition since the rollback will result in the materialized view being placed in the DELETE ONLY state.

  – An alternative method would be to temporarily make the materialized view look like a regular table so that it can be populated directly using LOAD[8] or suitably batched insert statements with subselects corresponding to the query used to define the materialized view. When the entire table is populated, convert this table back to an materialized view using the SET SUMMARY option in the ALTER TABLE statement as discussed in 2.5, “Materialized view ALTER considerations” on page 41.

  You can choose to increase the space for active logs via either of the following methods:

  – Increase the number of secondary log files.

  – Set the number of secondary log files to ‘-1’ which is interpreted as infinite log space. This does not imply additional disks as long as log archival is used. With log archival, the active log is migrated to tertiary storage. In the event of a rollback, any recovery requiring log data from the archived log may be take an extended amount of time. Refer to the *Data Recovery and High Availability Guide and Reference (DATA-RCVR)* for more details.

[8] The input to this LOAD may come from an EXPORT to a file.

Tip: Incremental refresh should be used to reduce the duration of the refresh process, and should be considered when one or more of the following conditions exist:

- The volume of updates to the base tables relative to size of the base tables is small.
- Duration of read only access to the base tables during a full refresh is unacceptable.
- Duration of unavailability of the materialized view during a full refresh is unacceptable.

2.3.2 Immediate refresh

This maintenance approach is used when the materialized view must be kept in sync with any changes in the base tables on which it has been defined are updated. Such materialized views are called REFRESH IMMEDIATE tables.

Important: Not all materialized views can be defined to be REFRESH IMMEDIATE. The principle behind what materialized view can be defined as REFRESH IMMEDIATE is governed by the ability to compute the changes to the materialized view from the delta changes to the base tables, and any other base tables involved. Refer to 2.11, “Materialized view limitations” on page 92 for details about these restrictions.

Attention: Materialized view optimization occurs for both static and dynamic SQL statements with REFRESH IMMEDIATE tables.

Example 2-6 shows an example of SQL for creating a refresh immediate materialized view.

Example 2-6 Creating a refresh immediate materialized view

```
CREATE SUMMARY TABLE dba.summary_sales
   AS (SELECT ...........)
   DATA INITIALLY DEFERRED
   REFRESH IMMEDIATE
```

Note: The REFRESH TABLE statement can be issued against a REFRESH IMMEDIATE materialized view — it is generally used for initially populating the materialized view.

REFRESH IMMEDIATE tables are synchronized with the base tables in the same unit of work as the changes (inserts, updates or deletes) to the base tables. Given the synchronous nature of the immediate refresh capability, the atomic requirement for the change propagation can have a negative impact on transactions updating the base tables.

An incremental update mechanism is used to synchronize a REFRESH IMMEDIATE materialized view whenever update activity occurs against a base table. The process involved is shown in Figure 2-5.

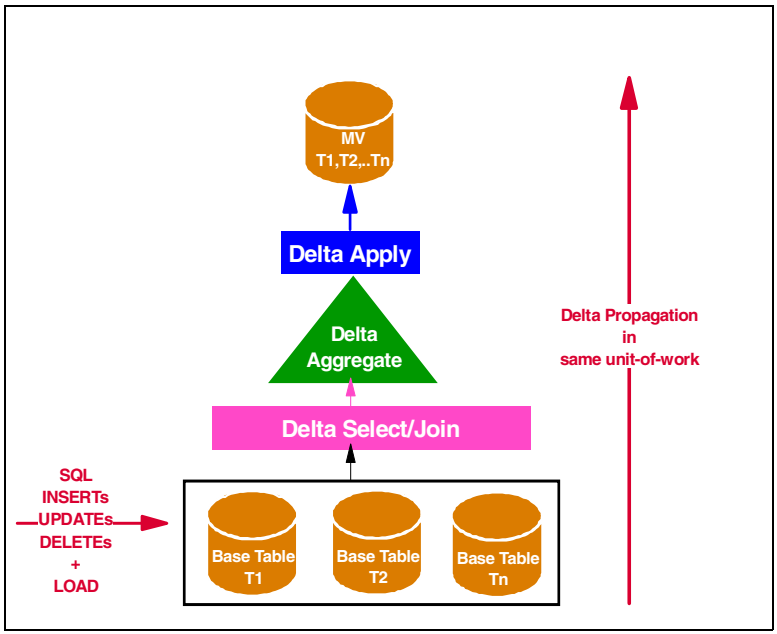


Figure 2-5 Immediate refresh using incremental update

When an SQL statement modifies a row in the base table, the following steps occur in atomic fashion:

1. The modified row is captured.
2. The query defining the materialized view is computed based on the modified row, computing the delta joins and delta aggregation to generate the data necessary to update the materialized query table.
3. The delta is applied to the materialized view.

Note: This processing occurs at statement execution time as opposed to occurring at commit time.

Consider, for example, that you have an materialized view that has data grouped by month. Assume that the data was up to and including the month of June. If a sales entry is made for the month of June, the delta change to the aggregation is computed so that the row in the materialized query table is updated to reflect the newly inserted row in the base sales table. If a row is inserted into the sales table for the month of July, a new row would be inserted in the materialized view since one did not exist before for July.

Note: DB2 may use pipelining or temporary tables to effect this operation.

Attention: Since this entire operation is atomic, any error encountered while updating either the base table or the materialized view will roll back all the changes during the unit of work. This guarantees the synchronization of the materialized view with the base tables.

When a LOAD INSERT operation is performed on the base table(s) of a materialized view, the newly appended data is invisible for read access, thus ensuring that the base table(s) and corresponding materialized views are still in sync.

In order to make the appended data visible and synchronized with the materialized view, perform the following steps:

1. Issue the following statement against the base table:

   ```
   SET INTEGRITY FOR base_table_name IMMEDIATE CHECKED
   ```

   This statement will make the base table available after verifying the consistency of the newly appended rows to the base table. It will also cause a CHECK PENDING NO ACCESS state to be set on the materialized view.

2. Issue a SET INTEGRITY or REFRESH TABLE of the materialized view with the INCREMENTAL option to bring the materialized view in sync with all the rows added by the LOAD INSERT operation.

See “Loading base tables (LOAD utility)” on page 37 for some of the options available when loading data into a base table.

Immediate refresh considerations

Two main considerations apply here as follows:

- REFRESH IMMEDIATE materialized views require updates to the underlying base table(s) to be reflected within the same unit of work. This atomic requirement can have a significant negative impact on transactions updating the base table(s). You should carefully evaluate the business requirements of

zero latency between the base tables and materialized view before choosing the REFRESH IMMEDIATE option.

- Another consideration is that lock contention may be an issue for higher level aggregation rows. The higher the level of aggregation, the greater the chance of concurrency issues against the materialized view when multiple users are concurrently performing updates on the base tables. For example, a materialized view containing the total sales for the year may need to update the row for the current active year for every sales transaction that pertains to the current year.

  One way to reduce contention on such REFRESH IMMEDIATE materialized views, is to redefine it as a REFRESH DEFERRED materialized view, and associate a staging table with it. An incremental refresh of this materialized view will significantly reduce this lock contention. However, since the semantics of data latency changes with this option, you should carefully evaluate the business requirements of zero latency between the base tables and materialized view before choosing this approach to reduce lock contention.

2.4 Loading base tables (LOAD utility)

The base tables of a materialized view may be updated either through SQL statements, or via the LOAD utility.

Attention: The following discussion applies to REFRESH IMMEDIATE materialized views and staging tables.

In DB2 V7, when a LOAD was performed on the base tables, *all* the corresponding materialized views were put in a CHECK PENDING NO ACCESS state, until complete synchronization of the materialized view and base tables had been accomplished via the SET INTEGRITY or REFRESH TABLE statements. When the tables involved are very large, the time to refresh may be very large, since the entire base data is scanned and not just the recently appended data. This can result in very poor response times for user queries, since materialized view optimization would be inhibited, because of its CHECK PENDING NO ACCESS state (see “Matching criteria considerations” on page 44 for details of limitations of matching).

Important: Remember that LOAD INSERT appends rows after the last page of the table.

In DB2 V8, functionality has been added to improve the availability of materialized views during a load of base tables as follows:

1. Reduce the time it takes to refresh a materialized view by only scanning the LOAD appended data. This applies to both refresh deferred and refresh immediate materialized views.
2. Keep the materialized views available for optimization during and after the load, by blocking access to the appended data, until all the (refresh immediate[9]) materialized views have been synchronized with the base tables.

In order to support this high availability functionality, the following capabilities have been added:

- Three new table states have been introduced:
  - CHECK PENDING READ ACCESS state:

    This allows read access to tables, but only up to and **not** including the first loaded page.
  - CHECK PENDING NO ACCESS state:

    This was previously just called the CHECK PENDING state, and had to be renamed given the new CHECK PENDING READ ACCESS state.
  - NO DATA MOVEMENT state:

    This ensures that the RID of a row cannot change. Therefore operations such as REORG, or REDISTRIBUTE, or update of a partitioning key, or an update of a key in an multi-dimensional cluster (MDC) table will all be inhibited. SQL insert, update (excepting those mentioned) and delete operations do not change the RIDs, and are therefore permitted.
- New options have been added to the LOAD:
  - CHECK PENDING CASCADE DEFERRED | IMMEDIATE:

    CASCADE DEFERRED specifies that descendent foreign key tables, and descendent refresh immediate and staging tables are **not** put into CHECK PENDING NO ACCESS state, but left in normal state.
  - ALLOW READ ACCESS | NO ACCESS:

    ALLOW READ ACCESS specifies that all the data prior to the first page appended can continue to be read, but not updated.
- A new option has been added to SET INTEGRITY:
  - FULL ACCESS:

[9] By definition, refresh deferred materialized views do not care about data latency issues.

This option specifies that full read write access should be allowed on the table, even if there are dependent materialized views for this table that have not yet been refreshed with the newly load appended data. If this option is not specified, the table has the NO DATA MOVEMENT mode set on it.

Assume a scenario shown in Figure 2-6. Here SALES is the base table on which two materialized views SALES_SUM and SALES_SUM_REGION are defined. The SALES table has check constraints in its definition (such as region code checking), and is also involved in referential integrity constraints with the PRODUCT and STORE_SALES tables.

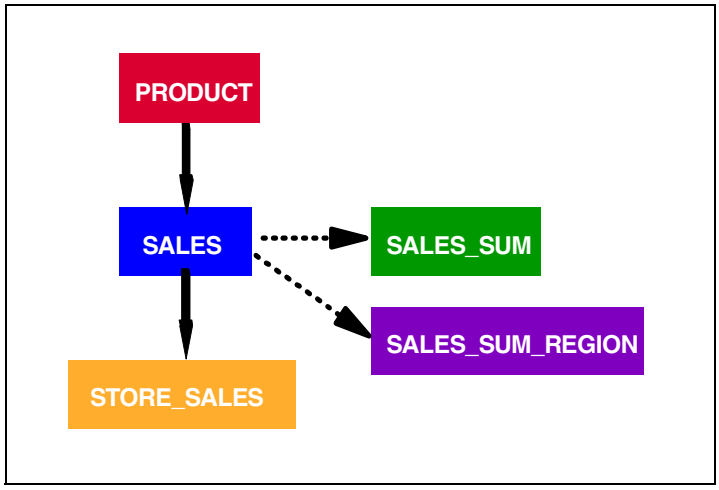


Figure 2-6 LOAD application sample

Assume a LOAD insert[10] is done on SALES with the CHECK PENDING CASCADE DEFERRED option.

```
LOAD INSERT INTO SALES ..... CHECK PENDING CASCADE DEFERRED...ALLOW READ
ACCESS...
```

1. LOAD issues a SET INTEGRITY SALES OFF which causes SALES to be put in CHECK PENDING READ ACCESS state (because of the ALLOW READ ACCESS option), and the data gets loaded.
2. STORE_SALES, PRODUCT, SALES_SUM and SALES_SUM_REGION are left in normal state because of the CHECK PENDING CASCADE DEFERRED option.

[10] This causes data to be appended to existing data in the table. If LOAD replace is done, the entire contents of the table are deleted and replaced by the new data. The CHECK PENDING CASCADE DEFERRED option can still be used for LOAD REPLACE, but the ALLOW READ ACCESS option cannot be used for LOAD REPLACE.

SQL statements will only be able to access the SALES table data **prior to the beginning of the first loaded page** (because of the ALLOW READ ACCESS option), and also be able to use SALES_SUM and SALES_SUM_REGION for optimization because they are still in sync. This has expanded the window of availability of the materialized view and base table.

3. At the end of the load, SALES still has the CHECK PENDING READ ACCESS state set.
4. Next[11] a SET INTEGRITY SALES ....IMMEDIATE CHECKED....is issued for verifying the integrity of the new data loaded. This takes an exclusive lock on SALES and puts it into a NO DATA MOVEMENT state, and also puts the SALES_SUM and SALES_SUM_REGION materialized views in a CHECK PENDING NO ACCESS state. STORE_SALES will remain in normal state since the rows added do not affect the referential integrity relationship with STORE_SALES. The PRODUCT table is also unaffected by the rows added to SALES and is therefore left in normal state.

   Note: SET INTEGRITY SALES will cause local check constraints to be verified, as well as referential integrity violations checked against the PRODUCT table. These checks may fail, which would result in states being rolled back to the way it was at the end of the load.

   At this point, materialized view optimization will be suspended because both the SALES_SUM and SALES_SUM_REGION materialized views are in CHECK PENDING NO ACCESS state.
5. Assuming a successful SET INTEGRITY SALES step, we issue a REFRESH TABLE SALES_SUM statement which results in an incremental update using only the data after the first loaded page, which is a much faster operation than scanning the entire base table. This reduces the window of availability of the SALES_SUM materialized view as well. When this refresh is completed successfully, the CHECK PENDING NO ACCESS state is reset on SALES_SUM, but not on SALES_SUM _REGION which has not been refreshed as yet. SALES continues to be in the NO DATA MOVEMENT state.

   Note: SALES_SUM materialized view is now available for optimization, while SALES_SUM_REGION is not, since it is in CHECK PENDING NO ACCESS state.

6. Now a REFRESH TABLE SALES_SUM_REGION causes that table to be taken out of CHECK PENDING NO ACCESS state and available for

[11] The SET INTEGRITY step is not required if the base table has no parent tables, descendent tables, check constraints, or generated columns.

materialized view optimization as well. Since this is the final materialized view on SALES, the NO DATA MOVEMENT state is reset on SALES.

The above features significantly improve the availability of the materialized views, and thereby the scalability and performance of queries.

Important: If the FULL ACCESS option is chosen on the SET INTEGRITY SALES step, then the NO DATA MOVEMENT state is **not** set on the SALES table. What this means is that full read/write access is permitted on SALES, and therefore incremental update is no longer possible on the SALES_SUM and SALES_SUM_REGION tables. When a REFRESH TABLE is issued against these tables, a full refresh is done which has a negative impact on availability of the materialized view. The decision to use FULL ACCESS is therefore an implementation choice.

2.5 Materialized view ALTER considerations

THE ALTER statement can be used to convert a materialized view to a regular table, and vice versa.

The following statement converts an existing materialized view into a regular table. This causes all the packages dependent on this materialized view to be invalidated.

```
ALTER TABLE tablename SET SUMMARY AS DEFINITION ONLY
```

The following statement converts an ordinary table into an materialized view, where the summary-table-definition defines the query and refreshable-table-options.

```
ALTER TABLE tablename SET SUMMARY AS summary-table-definition
```

ALTER may be used for several reasons:

- Correcting the materialized view options to address changing requirements over time.
- Temporarily taking materialized view optimization offline for maintenance, such as creating indexes.
- Taking it offline to avoid logging overhead as described in , “Deferred refresh considerations” on page 33.

The following restrictions apply to changing a regular table into an materialized view. The regular table must not:

- Already be a materialized view
- Be a typed table

- Have any constraints, unique indexes or triggers defined on it
- Be referenced in the definition of another materialized view

Also, you cannot ALTER a regular table into a staging table or vice versa.

2.6 Materialized view DROP considerations

When a materialized view is dropped, all dependencies are dropped and all packages with dependencies on the materialized view are invalidated. Views based on dropped materialized views are marked inoperative.

2.7 Materialized view matching considerations

The DB2 SQL Compiler analyzes user queries and produces an optimal access path to produce the desired results. These are the two key components most relevant to materialized views:

- Query rewrite component:

  This component analyzes the query and if appropriate, rewrites this query into another form that it believes will perform in superior fashion to the one written by the user. This capability frees the user from having to deal with different syntactic representations of the same semantic query, and instead focus on using syntax (s)he is most comfortable with.

  Part of this query rewrite process is the task of considering materialized views for optimization. This includes checking for certain:

  - States
  - Matching criteria

- Cost based optimizer component

  This component performs a cost analysis of materialized view processing versus base table access, and decides on the optimal access path.

Figure 2-7 graphically depicts this process.

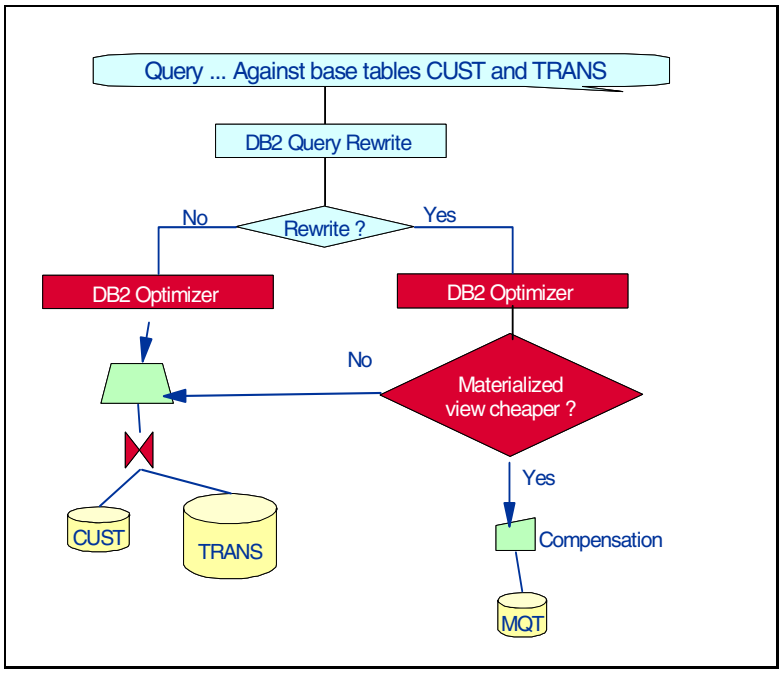


Figure 2-7 Materialized view optimization flow

We focus here on the query rewrite component task relating to materialized views, that is, state considerations and matching criteria considerations.

Important: DB2's EXPLAIN tables capture information about static and dynamic SQL statements, costs, and access path selected for a query. The EXPLAIN_STATEMENT table in particular, contains the text of the original SQL statement entered by the user, along with the rewritten (if appropriate) SQL statement used by the DB2 optimizer to choose the optimal access path for executing the SQL query. This potentially modified SQL statement may bear little resemblance to the original SQL query, as it may have been rewritten and/or enhanced with additional predicates as determined by the DB2 SQL Compiler.

DB2's Snapshot Monitor captures dynamic SQL statement executions in a statement cache and gathers statistics about them, such as the number of executions, number of rows reads and updated, and execution times. However, this statement cache only includes the user's original dynamic SQL query, and not the version that gets to the DB2 optimizer after query rewrite.

2.7.1 State considerations

The following state considerations apply for DB2 to even consider materialized view optimization:

- Materialized view must be created with the ENABLE QUERY OPTIMIZATION parameter.
- CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION register must enable optimization of the particular table type. This register can be set to:

  ```
  SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION = ALL|NONE|SYSTEM|USER
  ```

- For REFRESH DEFERRED materialized views, the CURRENT REFRESH AGE register must be set to ANY.

  ```
  SET CURRENT REFRESH AGE ANY|0
  ```

- REFRESH IMMEDIATE materialized views are always current and are always candidates for materialized view optimization, regardless of the CURRENT REFRESH AGE register setting.
- For dynamic and static SQL, the QUERY OPTIMIZATION level must be set to 2, or greater than equal to 5.
  - Default value is 5, and this default can be changed in the DFT_QUERYOPT parameter in the database configuration file
  - The level can be changed as follows

    ```
    SET CURRENT QUERY OPTIMIZATION LEVEL 2
    ```

- Materialized view cannot be in a CHECK PENDING NO ACCESS state.

2.7.2 Matching criteria considerations

Matching is the process of reviewing the user query, and evaluating the potential use of a materialized view for query rewrite.

Assuming that the state criteria are not inhibitors, the query rewrite component reviews the following criteria to determine the viability of using the materialized view in the query rewrite.

We discuss these criteria as:

- Matching permitted
- Matching inhibited

Quite often, a materialized view may not exactly match the user query, and DB2 may have to incur some extra processing to massage the materialized view data to deliver the desired result. This extra processing is called *compensation.* List item 2 on page 46 shows an example of compensation.

2.7.3 Matching permitted

Materialized views will be considered for optimization in the following cases:

1. **Superset predicates and perfect match:**

   This is the simplest case where the user query has the same number of tables as in the materialized view, and the same expressions, and requests an answer that can be fully met with the data in the materialized view. Here, the predicates involved in the materialized view must be a superset of those involved in the query. In DB2 V7, predicate analysis to detect this was limited, with only exact matches, or simple equality predicates and IN predicates being considered. In DB2 V8, the analysis has been expanded to cover a broader range of predicates.

   Scenario 1: Consider the materialized view shown in Example 2-7:

Example 2-7 Superset predicates and perfect match materialized view 1

```
CREATE SUMMARY TABLE custtrans AS
   (
   SELECT cust_id, COUNT(*) AS counttrans
   FROM trans
   GROUP BY cust_id
   )
DATA INTITIALLY DEFERRED REFRESH DEFERRED
```

   A query that looks like the one shown in Example 2-8 will be considered matching for materialized view optimization purposes.

Example 2-8 Superset predicates and perfect match — matching query 1

```
SELECT cust_id, COUNT(*)
FROM trans
WHERE cust_id  > 1000
GROUP BY cust_id
```

   Scenario 2: Consider the materialized view shown in Example 2-9:

Example 2-9 Superset predicates and perfect match materialized view 2

```
CREATE SUMMARY TABLE custtrans AS
   (
   SELECT cust_id, COUNT(*) AS counttrans
   FROM trans
   WHERE cust_id > 500
   GROUP BY cust_id
   )
DATA INTITIALLY DEFERRED REFRESH DEFERRED
```

A query that looks like the one shown in Example 2-10 will be considered matching for materialized view optimization purposes.

Example 2-10 Superset predicates and perfect match — matching query 2

```
SELECT cust_id, COUNT(*)
FROM trans
WHERE cust_id  > 1000
GROUP BY cust_id
```

Figure 2-8 provides additional examples of matching conditions. The “Valid ?” column indicates whether or not the materialized view will be considered for optimization or not for the given query.

QUERY	Materialized View	Valid ?
... cust_age >= 15 ...	... cust_age >= 20 ...	No
... cust_age >= 25 ...	... cust_age >= 20 ...	Yes
... trans_yr IN (2001, 2002) ...	... trans_yr IN (2000, 2001) ...	No
... trans_yr = 2002 ...	... trans_yr IN (2000, 2001, 2002) ...	Yes
SELECT date, cust_id, SUM(sales) FROM TRAN WHERE ...	SELECT cust_id, SUM(SALES) FROM TRAN WHERE ...	No
SELECT cust_id, INT((cust_age + 5) / 10), SUM(sales) FROM ... WHERE ...	SELECT cust_id, INT(cust_age / 10), SUM(sales) FROM ... WHERE ...	No

Figure 2-8 Matching columns, predicates, and expressions

2. **Aggregation functions and grouping columns:**

   Aggregation collapses related groups of rows, resulting in a smaller size of the materialized view. It is not necessary to define different materialized views for each type of user grouping. Under certain conditions, DB2 can decide to use a materialized view even if the materialized view’s grouping is different from that of the user query. For instance, if the materialized view has a GROUP BY on a finer granularity, DB2 can compute the result of a coarser granularity GROUP BY by doing further aggregation[12] on top of the materialized view as shown in the following discussion.

[12] This is also called compensation.

Scenario 1: Consider the materialized view shown in Example 2-11, which has one row for every month of every year.

Example 2-11 Aggregation functions & grouping columns materialized view 1

```
CREATE SUMMARY TABLE dba.trans_agg AS
   (
   SELECT ti.pgid, t.locid, t.acctid, t.status,
             YEAR(pdate) as year, MONTH(pdate) AS month,
             SUM(ti.amount) AS amount, COUNT(*) AS count
   FROM  transitem AS ti, trans AS t
   WHERE  ti.transid = t.transid
   GROUP BY YEAR(pdate), MONTH(pdate)
   )
DATA INITIALLY DEFERRED REFRESH IMMEDIATE
```

The query shown in Example 2-12, with a GROUP BY on YEAR, can be computed from the above materialized view by aggregating all the months of a year.

Example 2-12 Aggregation functions & grouping columns — matching query 1

```
SELECT ti.pgid, t.locid, t.acctid, t.status,
             YEAR(pdate) AS year, MONTH(pdate) AS month,
             SUM(ti.amount) AS amount, COUNT(*) AS count
FROM  transitem AS ti, trans AS t
WHERE  ti.transid = t.transid
GROUP BY YEAR(pdate)
```

This capability allows the DBA to optimize by only defining one materialized view at the month level. This is the simplest form of matching handled by DB2 as far as grouping columns are concerned. The number of materialized views can be minimized by using complex constructs that include grouping sets, ROLLUP and CUBE operators. Refer to 3.3.2, “GROUPING capabilities ROLLUP & CUBE” on page 125 for an overview of DB2’s support of complex GROUP BY constructs.

Following are some scenarios using grouping sets:

Scenario 2: Consider the materialized view shown in Example 2-13.

Example 2-13 Aggregation functions & grouping columns materialized view 2

```
CREATE SUMMARY TABLE AST1 AS
   (
   SELECT .....GROUP BY GROUPING SETS
      ((customer_id, product_group_id), YEAR(date_col), MONTH(date_col))
   )
```

```
DATA INITIALLY DEFERRED REFRESH IMMEDIATE
```

The DB2 query rewrite engine would consider matching any one of the queries shown in Example 2-14 against the prior materialized view, assuming there are no other inhibitors.

Example 2-14 Aggregation functions & grouping columns — matching query 2

```
SELECT ..... GROUP BY customer_id, product_group_id
SELECT ..... GROUP BY customer_id
SELECT ..... GROUP BY product_group_id
SELECT ..... GROUP BY YEAR(date_col)
SELECT ..... GROUP BY MONTH(date_col)
```

Scenario 3: Consider the materialized view shown in Example 2-15:

Example 2-15 Aggregation functions & grouping columns materialized view 3

```
CREATE SUMMARY TABLE ast AS
   (
   SELECT store_id, cust_id, year, month, COUNT(*) as cnt
   FROM Trans
   GROUP BY GROUPING SETS
      (
      (store_id, cust_id, year),
      (store_id, year),
      (store_id, year, month),
      (year)
      )
   )
DATA INITIALLY DEFERRED REFRESH IMMEDIATE
```

The query shown in Example 2-16 can be satisfied from the above materialized view by simply filtering out the rows pertaining to the other entities in the grouping set.

Example 2-16 Aggregation functions & grouping columns — matching query 3

```
SELECT store_id, year,  COUNT(*) as cnt
FROM Trans
WHERE year > 1990
GROUP BY store_id, year
```

Note: In the above case, assuming that columns *store_id* and *year* are defined as NOT NULL columns, the DB2 query rewrite engine transforms the query automatically internally, as shown in Example 2-17.

Example 2-17 Internally rewritten query by DB2 using the materialized view

```
SELECT store_id, year, cnt
FROM AST
WHERE store_id IS NOT NULL AND
              year IS NOT NULL AND
              cust_id IS NULL AND
              month IS NULL AND
              year > 1990
```

The following extra predicates are the only compensating predicates needed to make use of the materialized view.

- The two predicates store_id IS NOT NULL AND year IS NOT NULL ensure that all the rows from the (store_id, year) grouping are retrieved.
- The cust_id IS NULL predicate ensures that nothing will qualify from the (store_id, cust_id, year) grouping.
- The month IS NULL predicate will filter out the rows pertaining to the (store_id, year, month) grouping.
- The store_id IS NOT NULL predicate will ensure that nothing will qualify from the (year) grouping as well as the empty grouping ().

More complex scenarios might require further grouping.

Similar scenarios can be shown using the ROLLUP and CUBE operators to satisfy user queries having many different combinations of column groupings.

Important: Such complex materialized views involving grouping sets, and ROLLUP and CUBE operators not only save disk space, but also can save refresh times since a single scan of the base tables is sufficient to populate the materialized view. Creating multiple materialized views on the other hand, would require individual REFRESH statements to be executed for each materialized view, and thereby individual accesses against the base tables.

Attention: The assumption in the above scenarios is that the columns or expressions in the GROUP BY list are defined as NOT NULL. In the event that this is not true, that is, the GROUP BY list items are nullable, then the materialized view *must have* GROUPING identifiers defined as highlighted in Example 2-18, in order for the DB2 optimizer to consider matching the user query to the materialized view. The reason for this requirement is that the DB2 optimizer needs to distinguish between NULL values within a materialized view that could either be due to a rolled up aggregation involving missing rows, or due to actual NULL values in the base tables itself.

The GROUPING function *only* needs to be defined for all nullable columns when super aggregates (ROLLUP, CUBE and grouping sets) are involved.

Example 2-18 Nullable columns or expressions in GROUP BY

```
CREATE TABLE S2 AS
(SELECT Period.year,
       Product.id,
       Fact.DeliveryCode,
       GROUPING(Period.year) AS gpyear,
       GROUPING(Product.id) AS gpprodid,
       GROUPING(Fact.DeliveryCode) AS gpfactDC,
       SUM(Fact.Quantity) AS Quantity,
       SUM(Fact.Amount) AS Amount,
       SUM(Fact.Quantity * Product.Price) AS QP_Amount
FROM Fact,
   Product,
   Period
WHERE Fact.prod_id = Product.id and
     Fact.period_id = Period.id
GROUP BY rollup(Period.year,
            Product.id,
            Fact.DeliveryCode ))
DATA INITIALLY DEFERRED REFRESH DEFERRED
```

In this example, columns *Period.Year, Product.id* and *Fact.DeliveryCode* in the GROUP BY are nullable. Therefore, the materialized view will have rows with NULL values that may either be due to a rolled up aggregation (sub-total), or due to NULL values from the base table itself. The GROUPING identifier enables you to differentiate between these two cases. A value of one in this column indicates that the row was the result of a sub-total from the GROUP BY function, while a value of zero indicates otherwise. An example of the results of using the GROUPING identifier is shown in “GROUPING, GROUP BY and CUBE example” on page 138.

The query in Example 2-19 has a ROLLUP on fewer columns (only *Period.Year,* and *Product.id)* than in the materialized view.

Example 2-19 Nullable columns or expressions in GROUP BY — user query

```
SELECT Period.year,
       Product.id,
       SUM(Fact.Quantity) AS Quantity,
         SUM(Fact.Amount) AS Amount,
       SUM(Fact.Quantity * Product.Price) AS QP_Amount
FROM Fact,
   Product,
   Period
WHERE Fact.prod_id = Product.id and
     Fact.period_id = Period.id
GROUP BY rollup(Period.year,
          Product.id)
```

Assuming this query matches the materialized view in Example 2-18 on page 50, then the query would be rewritten as shown here in Example 2-20, to filter out the unwanted rows.

Example 2-20 Rewritten query

```
SELECT year,
       id,
       Quantity,
       Amount,
         QP_Amount
FROM S2
WHERE gpfactDC = 1
```

The ‘gpfactDC = 1’ predicate in the rewritten query only selects those rows in the materialized view returned by the GROUP BY function, and ignores those relating to NULLs in the base table itself.

Figure 2-9 provides additional examples of matching conditions. It shows that the GROUP BY expressions *must be* derivable from the materialized view.

QUERY	Materialized View	Valid ?
... GROUP BY store_id	... GROUP BY cust_id, store_id	Yes
... GROUP BY ((cust_age + 5) / 10)	... GROUP BY cust_age / 10	No
... CUBE(cust_id, store_id)	... GROUPING SETS (cust_id, store_id)	No
... GROUPING SETS (cust_id, store_id)	... CUBE(cust_id, store_id)	Yes
... AVG(sales) ...	... SUM(sales), COUNT(*) ...	Yes
... COUNT(DISTINCT cust_id) ...	... cust_id, COUNT(*)	No

Figure 2-9 Matching GROUP BY and aggregate functions

3. **Extra tables in the query:**

   DB2 is able to match user queries that contain more tables than those defined in the materialized view, when the join predicates to the base tables can be replaced by join predicates between the materialized view and the additional tables.

   Consider the materialized view shown in Example 2-21:

Example 2-21 Extra tables in the query materialized view

```
CREATE SUMMARY TABLE dba.trans_agg AS
   (
   SELECT ti.pgid, t.locid, t.acctid, t.status, YEAR(pdate) AS year,
      MONTH(pdate) AS month, SUM(ti.amount) AS amount, COUNT(*) AS count
   FROM transitem AS ti, trans AS t
   WHERE  ti.transid = t.transid
   GROUP BY YEAR(pdate), MONTH(pdate), ti.pgid, t.locid, t.acctid, t.status
   )
DATA INITIALLY DEFERRED REFRESH IMMEDIATE
```

   A user query that looks like the one shown in Example 2-22 can be considered matching the above materialized view even though it has an additional location table *loc* included.

Example 2-22 Extra tables in the query — matching query

```
SELECT YEAR(pdate) AS year, loc.country, SUM(ti.amount) AS amount, COUNT(*)
   AS count
FROM transitem AS ti, trans AS t, loc AS loc
WHERE  ti.transid = t.transid AND t.locid = loc.locid
      AND YEAR(pdate) BETWEEN 1990 and 1999
GROUP BY YEAR(pdate), loc.country
```

The *loc* is joined on the locid column to *trans*. The locid column is one of the GROUP BY columns of the materialized view. DB2 can use this column to join the relevant rows of the materialized view **after** applying the YEAR predicate with the *loc* table. The aggregated results can then be further consolidated by grouping on the YEAR(pdate) and country.

4. **Extra tables in the materialized view:**

   DB2 is able to match user queries against materialized views that have more tables than defined in the query, in certain cases where referential integrity is known to exist.

   Consider the materialized view shown in Example 2-23.

Example 2-23 Extra tables in the materialized view

```
CREATE TABLE dba.PG_SALESSUM  AS
   (
   SELECT l.lineid AS prodline, pg.pgid AS pgroup, loc.country, loc.state,
      YEAR(pdate) AS year, MONTH(pdate) AS month, SUM(ti.amount) AS amount,
      COUNT(*) AS count
   FROM transitem AS ti, trans AS t, loc AS loc,
      pgroup AS pg, prodline AS l
   WHERE  ti.transid = t.transid AND ti.pgid = pg.pgid
      AND pg.lineid =l.lineid AND t.locid = loc.locid
   GROUP BY  loc.country, loc.state, year(pdate),month(pdate), l.lineid,
      pg.pgid
   )
DATA INITIALLY DEFERRED REFRESH IMMEDIATE
```

A user query as shown in Example 2-24 can be considered as matching the above materialized view even though the materialized view has two more tables *pgroup* and *prodline* than in the user query:

Example 2-24 Extra tables in the materialized view — matching query

```
SELECT YEAR(pdate) AS year, loc.country,
       SUM(ti.amount) AS amount, COUNT(*) AS count
FROM transitem AS ti, trans AS t, loc AS loc
WHERE  ti.transid = t.transid   AND t.locid = loc.locid
      AND YEAR(pdate) BETWEEN 1990 and 1999
GROUP BY YEAR(pdate), loc.country
```

The query in Example 2-24 references three tables, while the materialized view in Example 2-23 has five. It would appear at first glance that these additional tables *pgroup* and *prodline* would affect the result of the above query, if that materialized view were used in the query rewrite. This would be true unless DB2 was aware of referential integrity relationships being involved. For instance, if the *pgroup* and *prodline* tables were related to the other tables through referential integrity, it will not affect the number of rows in the result. They could be considered as look-up tables that are merely adding columns to the output.

Important: Referential integrity may either be system maintained, or provided as informational constraints (see the NOT ENFORCED constraint attribute in Example 2-25). With informational referential integrity constraints, the onus is on the DBA to guarantee the integrity of reference, since DB2 makes no attempt to enforce referential integrity. Informational referential integrity constraints help the DB2 optimizer make superior decisions about matching user queries that have fewer tables than those defined in the materialized view.

Example 2-25 Informational and system-maintained referential integrity constraints

```
-- INFORMATIONAL REFERENTIAL INTEGRITY CONSTRAINT
CREATE TABLE transitem (
.....
.....
pgid INT,
....
CONSTRAINT fk_pgid FOREIGN KEY (pgid) REFERENCES pgroup
ON DELETE CASCADE NOT ENFORCED
.....

-- SYSTEM-MAINTAINED REFERENTIAL INTEGRITY CONSTRAINT
CREATE TABLE pgroup (
.....
.....
lineid INT,
....
CONSTRAINT fk_lineid FOREIGN KEY (lineid) REFERENCES prodline
ON DELETE CASCADE ENFORCED
```

System-maintained referential integrity (where the constraint attribute is ENFORCED), as well as informational referential integrity constraints let the query rewrite component know of the existence of referential integrity. In such cases, the additional tables in the materialized view are guaranteed not to add or remove rows in the result, and the query rewrite engine can proceed with the materialized view matching optimization, and ignore these tables.

In the above scenario, *transitem* table is joined to the *pgroup* table on column pgid. If *pgroup.pgid* is the primary key in the referential integrity relationship, every value of t*ransitem.pgid* has one and only one value in *pgroup*. Furthermore, if the *prodline* table has a referential integrity relationship with the *pgroup* table, where *prodline.lineid* is the primary key, this join is also a join that does not affect the number of rows in the output. The materialized view can now be used for applying the query predicate, selecting the columns required by the query, and consolidating the aggregation by further grouping on only the columns required in the query.

5. **CASE expressions in the query:**

   Typically, matching queries with complex expressions need to have these complex expressions used in a similar way in the materialized view. There are some common scenarios that DB2 will handle. For example, DB2 can match some user queries with CASE expressions to a materialized view that contains the elements of the CASE expression as part of the GROUP BY clause, and the SELECT list of the materialized view. DB2 is able to match some user queries with CASE expressions as follows.

   Consider the materialized view shown in Example 2-26.

Example 2-26 CASE expression materialized view

```
CREATE TABLE S1 AS
(SELECT
   Period.year,
   Product.id,
   Fact.DeliveryCode,
   SUM(Fact.Quantity) AS Quantity,
   SUM(Fact.Amount) AS Amount,
   SUM(Fact.Quantity * Product.Price) AS QP_Amount
FROM
   Fact, Product, Period
WHERE
   Fact.prod_id = Product.id and
   Fact.period_id = Period.id
GROUP BY
   Period.year,
   Product.id,
   Fact.DeliveryCode)
DATA INITIALLY DEFERRED REFRESH DEFERRED
```

   Note that this materialized view does not have a CASE expression defined.

   The query in Example 2-27 will be considered matching for materialized view optimization purposes. The information required by the user query could easily be computed from the materialized view as long as *DeliveryCode* is part of the GROUP BY items in the materialized view.

Example 2-27 CASE expression — matching query

```
SELECT Period.Year,
   SUM(Fact.Quantity) ,
   SUM(
      (CASE WHEN Fact.DeliveryCode = 'Y'
                THEN (Fact.Amount)
            WHEN Fact.DeliveryCode = 'N'
                THEN (Fact.Quantity * Product.Price)
            ELSE 0
      END)
      ) AS RevenueForecast
FROM
   Fact, Product, Period
WHERE
   Fact.prod_id = Product.id and
   Fact.period_id = Period.id and
   Period.Year = 2002
GROUP BY Period.year,Product.id,Fact.DeliveryCode
```

2.7.4 Matching inhibited

Query rewrite component currently does *not* consider materialized view optimization in the following cases.

1. **Query includes the following constructs:**

   A query that includes the following constructs will not be considered for materialized view query rewrite. This is not a comprehensive list. Also, some of these restrictions may be removed in future releases.

   – A base table in the materialized view is itself a target of an UPDATE. For example, when there is a trigger involved, and the query may select from the same base table that it also updates.

   – Recursion or other complex constructs.

   – Physical property functions like NODENUMBER.

   – Outer Join.

   – UNION.

   – XMLAGG.

   – Window aggregation functions. These are aggregate functions specified with the OVER clause.

2. **Materialized view missing columns that are in the query:**

   If the materialized view is missing columns that exist in the base tables, and the query references those columns, then the materialized view will be ignored for optimization.

   Consider the materialized view shown in Example 2-28.

Example 2-28 Materialized view contains fewer columns than in query

```
CREATE SUMMARY TABLE custtrans AS
   (
   SELECT cust_id, COUNT(*) AS counttrans
   FROM trans
   GROUP BY cust_id
   )
DATA INTITIALLY DEFERRED REFRESH DEFERRED
```

   A user query that looks like the following will result in the materialized view being ignored for optimization purposes. This is because the" trans_date" column has not been defined in the materialized view shown in Example 2-29.

Example 2-29 Materialized view contains fewer columns than in query — no match

```
SELECT cust_id, COUNT(*)
FROM trans
WHERE trans_date > '2002-01-01'
GROUP BY cust_id
```

3. **Materialized view contains more restrictive predicates than in the query:**

   A materialized view cannot be considered matching if it is missing rows required to satisfy the user query.

   Note: The predicates involved in the materialized view must be a superset of the of those involved in the query. In DB2 V7, predicate analysis to detect this was limited, where only exact matches or simple equality predicates and IN predicates were considered. In DB2 V8, the analysis has been expanded to cover a broader range of predicates.

   Consider the materialized view shown in Example 2-30:

Example 2-30 Materialized view with more restrictive predicates

```
CREATE SUMMARY TABLE custtrans AS
   (
   SELECT cust_id, COUNT(*) AS counttrans
   FROM trans
   WHERE cust_id > 500
```

```
   GROUP BY cust_id
   )
DATA INTITIALLY DEFERRED REFRESH DEFERRED
```

A user query that looks like the following will result in the materialized view being ignored for optimization purposes. That is, rows corresponding to cust_id between 400 and 500 are missing in the materialized view shown in Example 2-31.

Example 2-31 Materialized view with more restrictive predicates — no match

```
SELECT cust_id, COUNT(*)
FROM trans
WHERE cust_id >= 400
GROUP BY cust_id
```

4. **Query with an expression not derivable from materialized view:**

   Even if the expression used in the materialized view is not identical to that used in the query, it might be possible to derive the expression used in the query from that in the materialized view. However, it is possible for some "obvious" matching cases to be ignored by DB2 due to precision or other issues. These restrictions will eventually be handled in future.

   Consider the materialized view shown in Example 2-32:

Example 2-32 Query: expression not derivable from materialized view

```
CREATE summary table custtrans AS
   (
   SELECT cust_id, SUM(sale_price) AS total, COUNT(items) AS countitems
   FROM trans
   GROUP BY cust_id
   )
DATA INTITIALLY DEFERRED REFRESH DEFERRED
```

A user query that looks like that shown in Example 2-33 will result in the materialized view being ignored for optimization purposes, since the expression cannot be derived by DB2. However, it is possible for some "obvious" matching cases to be ignored by DB2, either due to precision issues, truncation issues, or a possibly not implemented as yet. Some of these restrictions may eventually be handled in the future.

Example 2-33 Query: expression not derivable from materialized view — no match

```
SELECT cust_id, SUM(sale_price * 0.15) / COUNT(items)
FROM trans
GROUP BY cust_id
```

5. **Friendly arithmetic:**

   The database configuration parameter:

   ```
   DFT_SQLMATHWARN NO |YES
   ```

   sets the default value that determines the handling of arithmetic errors and retrieval conversion errors as errors (unfriendly) or warnings (friendly) during SQL statement compilation. For static SQL statements, the value of this parameter is associated with the package at BIND time. For dynamic SQL statements, the value of this parameter is used when the statement is prepared.

   The default is NO (unfriendly).

   Note: It is rare for this option to be changed after initial database creation, since the ramifications may be significant. Please refer to the *DB2 Administration Guide* for more details.

   The materialized view will **not** be considered for query rewrite if the query demands unfriendly arithmetic, and the materialized view supports friendly arithmetic.

   The materialized view will be considered for query rewrite when the query and materialized view have identical arithmetic requirements, and also when the query demands friendly arithmetic and the materialized view supports unfriendly arithmetic.

6. **Isolation mismatch**

   The isolation level of the materialized view must be equivalent or higher than that demanded of the user query.

   For example, if the materialized view is defined with ISOLATION of CS, then a query that requests:

   – Either UR or CS can match with the materialized view

   – RS or RR will not be considered for matching

   Note: It is important to know the ISOLATION under which the materialized view was created. The CLP command CHANGE ISOLATION TO... may be used to set the ISOLATION level before creating the materialized view.

2.8 Materialized view design considerations

Materialized views have the potential to provide significant performance enhancements to certain types of queries, and should be a key tuning option in every DBA's arsenal.

However, materialized views do have certain overheads which should be carefully considered when designing materialized views. These include:

- Disk space due to the materialized view and associated indexes, as well as staging tables.
- Locking contention on the materialized view during a refresh.
  - With deferred refresh, the materialized view is offline while the REFRESH TABLE is executing.
    - The same applies to staging table if one exists. Update activity against base tables is impacted during the refresh window.
  - With immediate refresh, there is contention on the materialized view when aggregation is involved due to SQL insert, update and delete activity on the base table by multiple transactions.
- Logging overhead during refresh of very large tables.
- Logging associated with staging tables.
- Response time overhead on SQL updating the base tables when immediate refresh and staging tables are involved, because of the synchronous nature of this operation.

Important: The objective should be to minimize the number of materialized views required by defining sufficiently granular REFRESH IMMEDIATE and REFRESH DEFERRED materialized views that deliver the desired performance, while minimizing their overheads.

When a materialized view has many tables and columns in it, it is sometimes referred to as a "wide" materialized view. Such a materialized view allows a larger portion of a user query to be matched, and hence provides better performance. However, when the query has fewer tables in it than in the materialized view, we need to have declarative or informational referential integrity constraints defined between certain tables in order for DB2 to use the materialized view for the query as discussed in 3., "Extra tables in the query:" on page 52.Note that a potential disadvantage of "wide" materialized views is that they not only tend to consume more disk space, but may also not be chosen for optimization because of the increased costs of accessing them.

When a materialized view has fewer columns and/or tables, it is sometimes referred to as a “thin” materialized view. In such cases, we reduce space consumption at the cost of performing joins during the execution of the query. For example, we may want to only store aggregate information from a fact table (in a star schema) in the materialized view, and pick up dimension information from the dimension tables through a join. Note that in order for DB2 to use such a materialized view, the join columns to the dimension tables must be defined in the materialized view. Note also that referential integrity constraints requirements do not apply to “thin” materialized views.

Attention: We will *only* be focusing on designing REFRESH DEFERRED materialized views, since the data warehouse environment is the predominant opportunity for exploiting materialized view optimization, and the data warehouse environment tends to overwhelmingly require access to other than current data.

Figure 2-10 provides an overview of the steps involved in designing REFRESH DEFERRED materialized views.

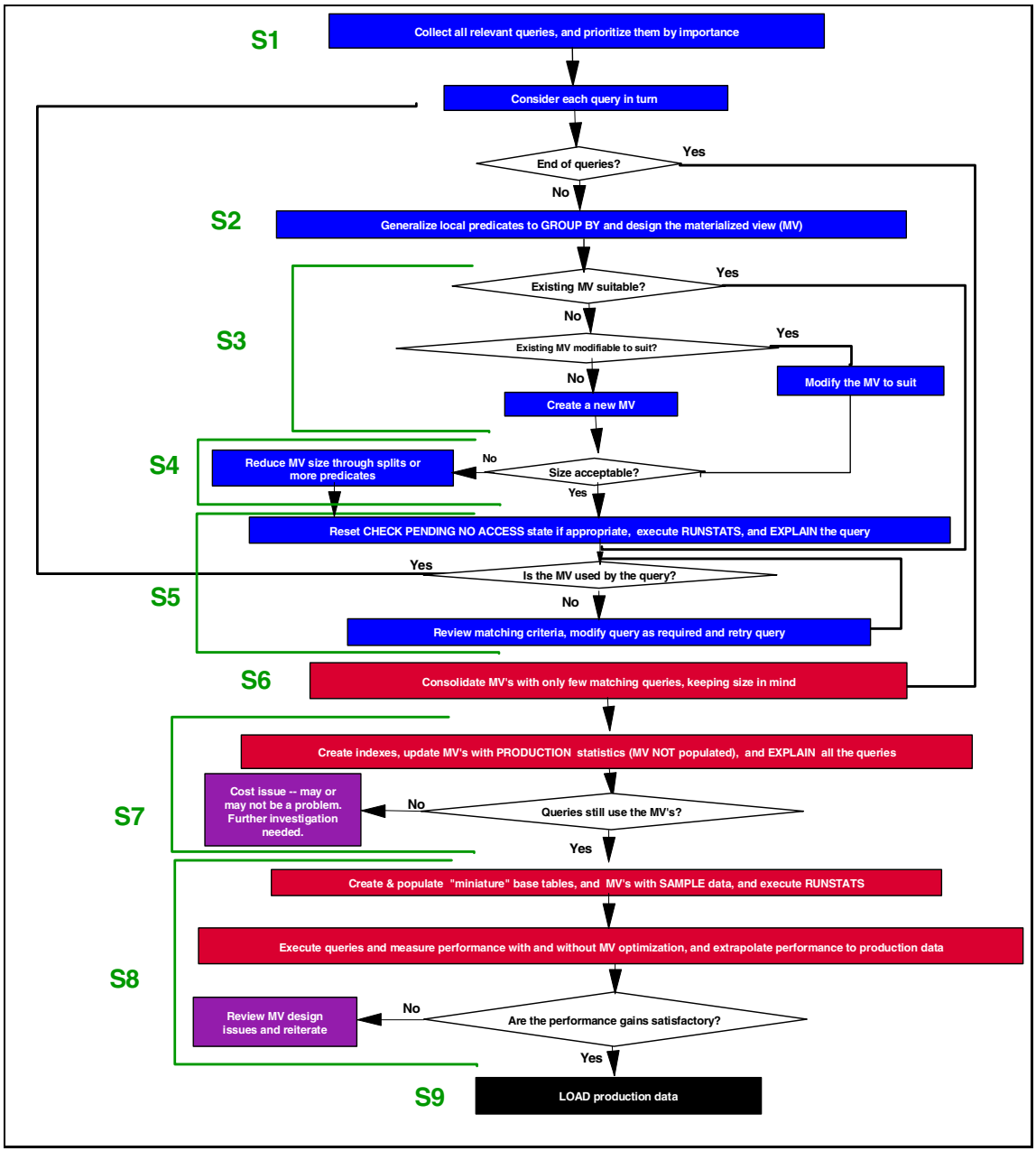


Figure 2-10 Overview of the design of REFRESH DEFERRED materialized views

We will briefly review each of these steps, and then use an application to illustrate generalizing of a few local predicates such as DISTINCT, compound, ROLLUP and CASE.

Note: We will only be focusing on dynamic SQL and deferred refresh materialized views.

2.8.1 Step 1: Collect queries & prioritize

The best source of information about SQL queries is the DB2 UDB package cache using the DB2 Snapshot Monitor. It provides a list of all the SQL statements, and the frequency of their execution.

The following command provides a verbose listing of all SQL statements, as well as the total number of executions per statement:

```
db2 get snapshot for dynamic sql on <database>
```

Figure 2-11 lists a portion of the results of this command.

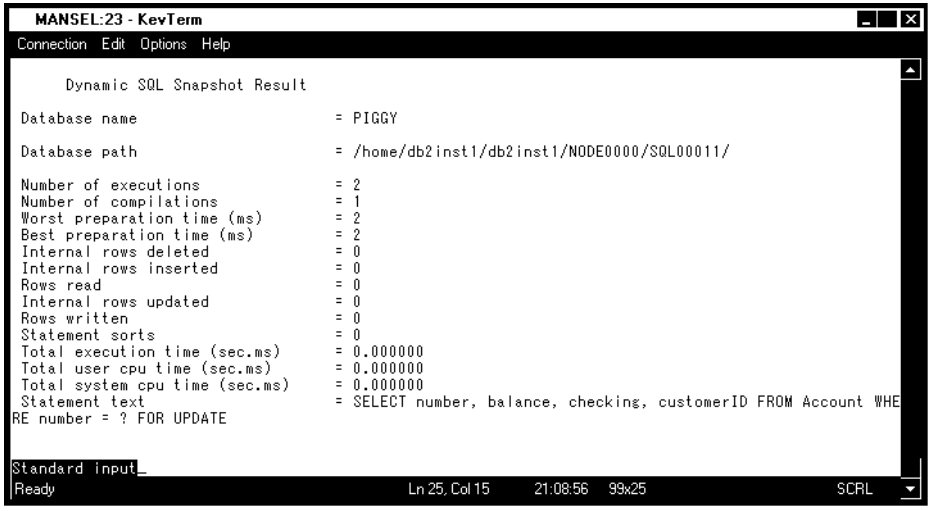


Figure 2-11 Get snapshot for dynamic SQL

Important: This approach only lists those SQL statements that are currently in the package cache. It excludes those statements that have been flushed since the last database restart or activation, as well as due to package cache size limitations. Users should therefore collect dynamic SQL statement executions over a period of time in order to arrive at the list of queries requiring optimization.

The advantage of extracting this information from the package cache is that it is less disruptive than running Event Monitor.

Once all the dynamic SQL statements of interest have been collected, they need to be prioritized by importance.

The SQL shown in Example 2-34 can be adapted to store the results of dynamic SQL capture into a suitable table for subsequent analysis.

Example 2-34 Capturing snapshot data into a table

```
connect to SAMPLE
delete from ADVISE_WORKLOAD where WORKLOAD_NAME='GET SNAPSHOT'
get snapshot for dynamic sql on SAMPLE write to file
insert into ADVISE_WORKLOAD (WORKLOAD_NAME, STATEMENT_TEXT, FREQUENCY, WEIGHT)
(select 'GET SNAPSHOT',stmt_text,NUM_EXECUTIONS, 1.0 from table
(SYSFUN.SQLCACHE_SNAPSHOT()) sqlsnap where stmt_text not like '%ADVISE_%' and
stmt_text not like '%EXPLAIN_%' and stmt_text not like '%SYSIBM.%' and
stmt_text not like '%SysCat.%' and stmt_text not like '%SYSCAT.%')
commit
connect reset
```

2.8.2 Step 2: Generalize local predicates to GROUP BY

Consider each query in turn, and generalize the local predicates to a GROUP BY. A very simple example of this exercise is shown in Example 2-35 and Example 2-36.

Example 2-35 Query involving a simple predicate

```
SELECT cust_id, COUNT(*)
FROM trans
WHERE cust_id  > 1000 AND cust_age < 50
GROUP BY cust_id
```

In Example 2-35, the simple predicate is "...WHERE custid > 1000 AND cust_age < 50". Generalizing the local predicate involves converting this predicate to a GROUP BY in a materialized view as shown in Example 2-36. The assumption made in this materialized view is that user queries are equally likely to choose from all possible values of *cust_id* and *cust_age,* since we have chosen not to add a filtering predicate in the materialized view.

In Example 2-9 on page 45, the materialized view has a filtering predicate of "...WHERE cust_id > 500" which implies that the predominant queries will choose *cust_id* values above 500, which might be associated with premium customers.

The decision to use filtering predicates should be based on domain expertise, and the frequency of query requests.

More detailed examples of generalizing local predicates are described in “Generalizing local predicates application example” on page 69.

Example 2-36 Generalize simple predicate to GROUP BY in a materialized view

```
CREATE SUMMARY TABLE custtrans AS
   (
   SELECT cust_id, cust_age, COUNT(*) AS counttrans
   FROM trans
   GROUP BY cust_id, cust_age
   )
DATA INTITIALLY DEFERRED REFRESH DEFERRED
```

2.8.3 Step 3: Create the materialized view

This step involves determining whether the query under consideration can reuse an existing materialized view “as is” or with some modifications, or require the creation of a new materialize view altogether.

In case an existing materialized view can be reused without modifications, then we need to proceed to bypass size estimation, and check whether the query under consideration actually routes to the materialized view. This is discussed in Step 5: Verify query routes to “empty” the materialized view.

2.8.4 Step 4: Estimate materialized view size

A key considerations in the design of materialized views is its size. Typically, the materialized view should be less than an order of magnitude in size as compared with the data in the base table(s). Large materialized views impact the cost formulas for access which might result in their being ignored for routing even though matching criteria are met. Large materialized views occupy more disk space, and potentially increase refresh cycle duration, thereby reducing availability of the materialized view.

If an estimate of the materialized view exceed this order of magnitude threshold[13], then this materialized view needs to be reduced in size by splitting it into two or more materialized views, and/or by adding filtering predicates to reduce the number of rows.

[13] Order of magnitude is a rule-of-thumb number — you can choose to revise this number upward or downward based on personal experience.

2.8.5 Step 5: Verify query routes to “empty” the materialized view

Having satisfied ourselves that the size of the materialized view is not an issue, we need to determine that the query will route to this materialized view. Since the routing is determined by matching criteria and materialized view access cost considerations, we need to eliminate the cost aspect of this routing consideration.

The following steps will verify whether the query can route to the materialized view based on matching criteria alone:

1. Create the materialized view
2. Remove it from the CHECK PENDING NO ACCESS state via the following command:

   ```
   SET INTEGRITY FOR tablename ALL IMMEDIATE UNCHECKED
   ```

3. Run **runstats** on the empty materialized view
4. Run EXPLAIN on the query, and verify that the query is being rewritten by the DB2 optimizer to route to the materialized view.

If the user query is being routed to the materialized view, then we can be confident that matching criteria are being met. However, we cannot be certain that routing will occur when the materialized view is populated with data from the base tables, since DB2 will weigh cost issues of materialized view access versus base table access in order to come up with an optimal access plan. Cost issues will be evaluated in Step 7: Introduce cost issues into materialized view routing.

Attention: If EXPLAIN indicates that the user query is *not* being routed to the materialized view, then the problem is with the matching criteria. The user query and/or the materialized view definition must be modified based on matching criteria discussed in “Matching criteria considerations” on page 44, and re-EXPLAINed to determine successful routing to the materialized view.

2.8.6 Step 6: Consolidate materialized views

Once all the queries have been processed, we need to try and minimize the number of materialized views. One approach is to review those materialized views that only have a few user queries routing to them, and try and consolidate them into a few number.

Note: Such consolidation efforts should keep size considerations in mind, while ensuring that all affected user queries continue to be routed to the appropriate consolidated materialized views.

2.8.7 Step 7: Introduce cost issues into materialized view routing

Once all the user queries have been processed and confirmed to route to the empty materialized views, we need to confirm that this routing will occur with populated materialized views as well.

Create appropriate indexes on the materialized views using the Index Advisor if needed, and update production data statistics for the materialized views and indexes.

Attention: Do *not* populate the materialized views with production data, as this may be a very time consuming process. We first need to verify routing with production data statistics before populating the materialized views.

If EXPLAIN shows that a particular query is *no longer* being routed to the materialized view, then it is because the DB2 optimizer has determined that it is more efficient to access the base tables directly than via the materialized view.

Important: This may or may not be a problem, since only an actual runtime comparison with and without materialized view routing can help pinpoint a potential issue. Forcing a routing to the materialized view will require updating materialized view statistics to deceive the DB2 optimizer into thinking that the materialized view has fewer rows than is actually the case. This scenario will have to be dealt on a case by case basis, depending upon the priority and performance of the query.

2.8.8 Step 8: Estimate performance gains

Having established that routing occurs to the materialized views with production statistics, we still need to determine whether such routing will result in satisfactory performance gains. However, given the sheer volume of data involved and the time it takes to load a materialized view, we need to ascertain performance gains without having to load the actual production data.

The solution is to:

1. Create “miniature” base tables and materialized views using sample data that is representative of production data.

   Important: Without a representative sample of the real world environment, performance estimates using this approach will be inconclusive,

2. Perform a comparison of query performance with and without materialized view routing.

The following is an overview of the steps involved:

1. Create “miniature” replicas of the base tables, materialized views, and their corresponding indexes.
2. Extract a representative sample of production data from the base tables, and populate the “miniature” base tables with this sample data.
3. Refresh the “miniature” materialized views from the “miniature” base tables.
4. Perform **runstats** against both “miniature” base tables and materialized views.
5. EXPLAIN the query to ensure that routing to the materialized views is occurring.

   Note: If routing does not occur, then the causes need to be investigated and corrected using techniques described earlier.

6. Execute the query and measure the performance with routing in effect.
7. Disable query optimization against the materialized view. This can be achieved either by setting the query optimization level as described in “State considerations” on page 44, or ALTERing the materialized view AS DEFINITION ONLY as described in “Materialized view ALTER considerations” on page 41.
8. EXPLAIN the query to ensure that the materialized view is being ignored. Take appropriate action to disable materialized view routing — worst case is to drop the materialized view!
9. Execute the query and measure the performance with *no* routing in effect.
10. Quantify the performance gains with and without routing to the materialized view.
11. Extrapolate the performance gains to production data.

Attention: If the estimated performance gains are unsatisfactory, then the design of the materialized view has to be reviewed, and the entire design process reiterated.

With satisfactory estimates of performance gains, we can proceed to the next step of loading the production data into the materialized views, executing **runstats** against them, and enabling them for optimization.

2.8.9 Step 9: Load the materialized views with production data

This step involves loading the materialized views with production data (via REFRESH TABLE statements), and making them available for DB2 optimization, after executing **`runstats`** against the materialized views and their corresponding indexes.

Important: The aforementioned steps describe a process that requires skilled professionals using trial and error techniques in order to design effective materialized views, and drop them when they are no longer beneficial. The process is both time consuming and error prone.

Attention: There are plans to provide an Design Advisor in future to assist DBAs define materialized views that deliver maximum performance benefits with minimal overheads.

2.8.10 Generalizing local predicates application example

Sapient is our sample application that explores data cubes with a star schema, by slicing and dicing multidimensional data. Any multidimensional data can be analyzed by this tool.

Sapient consists of a report view, and navigational controls.

- The report view allows for the viewing of the results of data queries on a data cube. Reports may be summary tables, trend line graphs or pie charts, etc.
- An important part of the navigational controls are the dimensions and metrics selection boxes. The dimension selection box allows the selection and drill down on each dimension. This includes drilling down a dimension hierarchy or cross drilling from one dimension to another. The metric selection box allows for the selection of the metrics that are computable for the given data cube. Additional navigation buttons allow forward and backward navigation to view previous reports, and the drill button to initiate the query to drill a hierarchy or cross drill a dimension.

We use the Medline data star schema shown in Figure 2-12 as the target of the Sapient application.

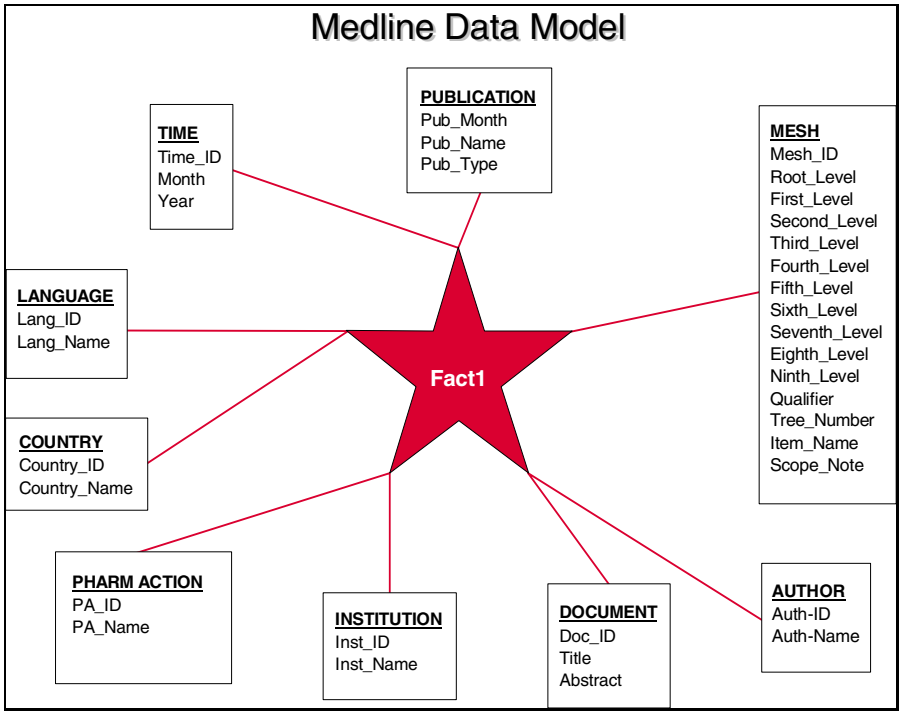


Figure 2-12 Sapient star schema

The Sapient application provides an interactive interface as shown in Figure 2-13 for accessing the contents of this star schema, and generates appropriate SQL queries to deliver the desired result.

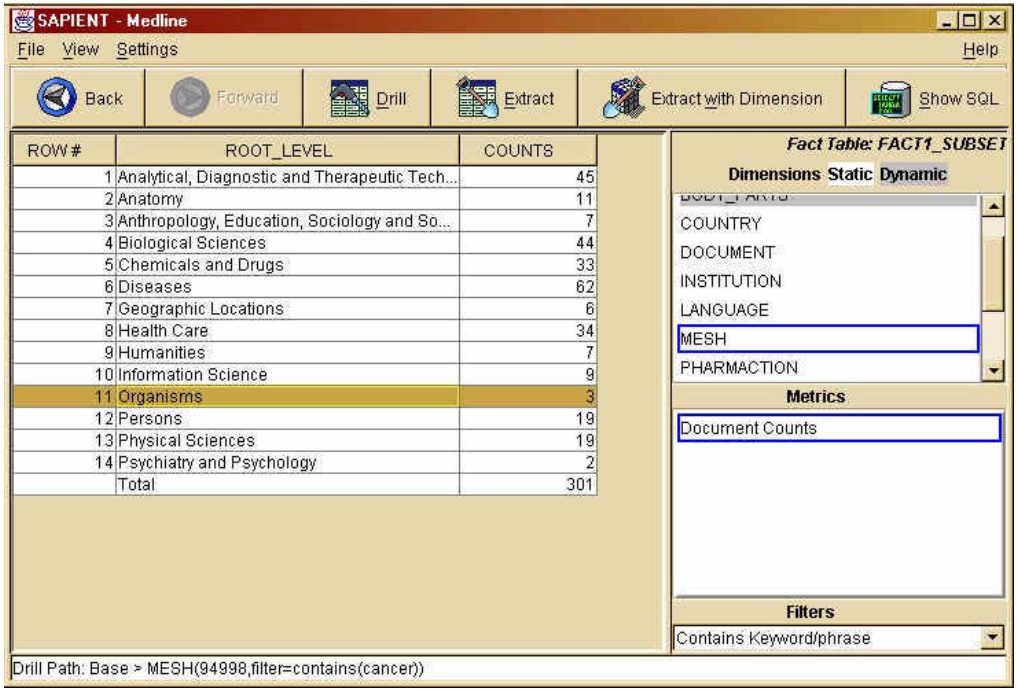


Figure 2-13 Sapient graphical user interface

For the purposes of our scenario, we assume that the performance of the generated queries are unsatisfactory, and would like to consider designing materialized views to improve their performance. Example 2-37 shows a list of the “problem” queries listed in priority order.

Example 2-37 “Problem” queries listed in priority order

```
Query 1:
SELECT MESH.SECOND_LEVEL, COUNT(DISTINCT FACT1_SUBSET.DOC_ID) COUNTS
FROM FACT1_SUBSET, MESH
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND MESH.ROOT_LEVEL='Anatomy'
   AND MESH.FIRST_LEVEL='Body Regions'
GROUP BY MESH.SECOND_LEVEL

Query 2:
SELECT MESH.FIRST_LEVEL, COUNT(DISTINCT FACT1_SUBSET.DOC_ID) COUNTS
FROM FACT1_SUBSET, MESH
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
   MESH.ROOT_LEVEL='Chemicals and Drugs'
GROUP BY MESH.FIRST_LEVEL

Query 3:
```

```
SELECT MESH.ROOT_LEVEL, COUNT(DISTINCT FACT1_SUBSET.DOC_ID) COUNTS
FROM FACT1_SUBSET, MESH
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID
GROUP BY MESH.ROOT_LEVEL
```

Query 4:

```
SELECT AUTHOR.AUTHOR_NAME, COUNT(DISTINCT FACT1_SUBSET.DOC_ID) COUNTS
FROM FACT1_SUBSET, MESH, AUTHOR
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
   FACT1_SUBSET.AUTHOR_ID=AUTHOR.AUTHOR_ID AND
   MESH.ROOT_LEVEL='Anatomy' AND MESH.FIRST_LEVEL='Animal Structures'
GROUP BY AUTHOR.AUTHOR_NAME
```

Query 5:

```
SELECT AUTHOR.AUTHOR_NAME, COUNT(DISTINCT FACT1_SUBSET.DOC_ID) COUNTS
FROM FACT1_SUBSET, MESH, AUTHOR
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
   FACT1_SUBSET.AUTHOR_ID=AUTHOR.AUTHOR_ID AND
   MESH.ROOT_LEVEL='Anatomy' AND MESH.FIRST_LEVEL IN ('Body Regions','Cells')
GROUP BY AUTHOR.AUTHOR_NAME
```

Query 6:

```
WITH DT AS
   (
   SELECT COUNTRY_NAME, COUNT(*) AS COUNT, YEAR, MONTH,
      ROOT_LEVEL, FIRST_LEVEL
   FROM FACT1_SUBSET, MESH, TIME, COUNTRY
   WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
      FACT1_SUBSET.TIME_ID=TIME.TIME_ID AND
      FACT1_SUBSET.COUNTRY_ID=COUNTRY.COUNTRY_ID
   GROUP BY ROLLUP(YEAR, MONTH), ROLLUP(ROOT_LEVEL, FIRST_LEVEL), COUNTRY_NAME
   )
SELECT *
FROM DT
ORDER BY COUNTRY_NAME, YEAR, MONTH, ROOT_LEVEL, FIRST_LEVEL
```

Query 7:

```
WITH DT AS
   (
   SELECT COUNTRY_NAME, COUNT(*) AS COUNT, YEAR, ROOT_LEVEL,
   FROM FACT1_SUBSET, MESH, TIME, COUNTRY
   WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
      FACT1_SUBSET.TIME_ID=TIME.TIME_ID AND
      FACT1_SUBSET.COUNTRY_ID=COUNTRY.COUNTRY_ID
   GROUP BY ROLLUP(YEAR), ROLLUP(ROOT_LEVEL), COUNTRY_NAME
   )
SELECT *
FROM DT
ORDER BY COUNTRY_NAME, YEAR, ROOT_LEVEL
```

```
Query 8:
SELECT AUTHOR.AUTHOR_NAME, DOC_ID,
   CASE WHEN
      FACT1_SUBSET.DOC_ID=1000 THEN COUNT(*)
      ELSE NULL END
FROM FACT1_SUBSET, MESH, AUTHOR
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
      FACT1_SUBSET.AUTHOR_ID=AUTHOR.AUTHOR_ID AND
      MESH.ROOT_LEVEL='Anatomy' AND
      MESH.FIRST_LEVEL IN ('Body Regions','Cells')
GROUP BY AUTHOR.AUTHOR_NAME, DOC_ID
```

We will consider each query in Example 2-37 in turn, and generalize the local predicates to design a materialized view, and then verify successful routing to this materialized view using EXPLAIN.

Query 1:

The materialized view for this query looks like AST3 in Example 2-38. Note the following generalization of local predicates:

- The predicates on columns MESH.ROOT_LEVEL and MESH.FIRST_LEVEL are added to the GROUP BY list, and removed from the predicates.
- The COUNT(DISTINCT FACT1_SUBSET.DOC_ID) select list item is replaced by a column DOC_ID, and an addition of column DOC_ID to the GROUP BY list.

Example 2-38 Materialized view AST3

```
CREATE SUMMARY TABLE AST3 AS
   (
   SELECT  ROOT_LEVEL, MESH.FIRST_LEVEL, SECOND_LEVEL, DOC_ID) COUNTS
   FROM  FACT1_SUBSET,  MESH WHERE  FACT1_SUBSET.MESH_ID= MESH.MESH_ID
   GROUP BY FIRST_LEVEL, ROOT_LEVEL, SECOND_LEVEL, DOC_ID
   )
DATA INITIALLY DEFERRED REFRESH DEFERRED  IN USERSPACE1
```

An estimate of the size of the materialized view was well within the limit of an order of magnitude as compared to the base table. After creating this materialized view, and populating it, we ran an EXPLAIN of Query 1 to confirm that it was being routed to AST3 as shown in Example 2-39.

Example 2-39 EXPLAIN of Query 1

```
******************** EXPLAIN INSTANCE ********************
DB2_VERSION: 07.02.0
SOURCE_NAME: SQLC2D01
```

```
SOURCE_SCHEMA: NULLID
EXPLAIN_TIME: 2002-08-26-16.47.17.125000
EXPLAIN_REQUESTER: DB2ADMIN

Database Context:
----------------
   Parallelism: None
   CPU Speed: 7.478784e-007
   Comm Speed: 0
   Buffer Pool size: 5256
   Sort Heap size: 20000
   Database Heap size: 600
   Lock List size: 50
   Maximum Lock List: 22
   Average Applications: 1
   Locks Available: 1243

Package Context:
---------------
   SQL Type: Dynamic
   Optimization Level: 5
   Blocking: Block All Cursors
   Isolation Level: Cursor Stability

---------------- STATEMENT 1  SECTION 203 ----------------
   QUERYNO: 1
   QUERYTAG:
   Statement Type: Select
   Updatable: No
   Deletable: No
   Query Degree: 1

Original Statement:
------------------
SELECT MESH.SECOND_LEVEL, COUNT(DISTINCT FACT1_SUBSET.DOC_ID) COUNTS
FROM FACT1_SUBSET, MESH
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND MESH.ROOT_LEVEL='Anatomy' AND
        MESH.FIRST_LEVEL='Body Regions'
GROUP BY MESH.SECOND_LEVEL

Optimized Statement:
-------------------
SELECT Q3.$C1 AS "SECOND_LEVEL", Q3.$C0 AS "COUNTS"
FROM
   (SELECT COUNT(Q2.$C1), Q2.$C0
   FROM
      (SELECT Q1.SECOND_LEVEL, Q1.COUNTS
      FROM DB2ADMIN.AST3 AS Q1
      WHERE (Q1.FIRST_LEVEL = 'Body Regions') AND (Q1.ROOT_LEVEL =
```

```
         'Anatomy')) AS Q2
   GROUP BY Q2.$C0) AS Q3
```

Query 2:

Note the following generalization of local predicates for this query:

- The predicate on column MESH.ROOT_LEVEL is added to the GROUP BY list, and removed from the predicate. MESH.ROOT_LEVEL is not a nullable column.
- The COUNT(DISTINCT FACT1_SUBSET.DOC_ID) select list item is replaced by a column DOC_ID, and an addition of column DOC_ID to the GROUP BY list.

We see that a materialized view from the above generalization is a coarser granular version of AST3, since it only groups on MESH.ROOT_LEVEL. We can therefore potentially reuse AST3.

We ran an EXPLAIN of Query 2 to confirm that it was being routed to AST3 as shown in Example 2-40.

Example 2-40 EXPLAIN of Query 2

```
******************** EXPLAIN INSTANCE ********************

DB2_VERSION: 07.02.0
SOURCE_NAME: SQLC2D01
SOURCE_SCHEMA: NULLID
EXPLAIN_TIME: 2002-08-26-16.58.54.265000
EXPLAIN_REQUESTER: DB2ADMIN

Database Context:
----------------
   Parallelism: None
   CPU Speed: 7.478784e-007
   Comm Speed: 0
   Buffer Pool size: 5256
   Sort Heap size: 20000
   Database Heap size: 600
   Lock List size: 50
   Maximum Lock List: 22
   Average Applications: 1
   Locks Available: 1243

Package Context:
---------------
   SQL Type: Dynamic
   Optimization Level: 5
```

```
   Blocking: Block All Cursors
   Isolation Level: Cursor Stability


---------------- STATEMENT 1  SECTION 203 ----------------
   QUERYNO: 1
   QUERYTAG:
   Statement Type: Select
   Updatable: No
   Deletable: No
   Query Degree: 1

Original Statement:
------------------
SELECT MESH.FIRST_LEVEL, COUNT(DISTINCT FACT1_SUBSET.DOC_ID) COUNTS
FROM FACT1_SUBSET, MESH
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND MESH.ROOT_LEVEL='Chemicals and
        Drugs'
GROUP BY MESH.FIRST_LEVEL


Optimized Statement:
-------------------
SELECT Q3.$C1 AS "FIRST_LEVEL", Q3.$C0 AS "COUNTS"
FROM
   (SELECT COUNT(Q2.$C1), Q2.$C0
   FROM
      (SELECT Q1.FIRST_LEVEL, Q1.COUNTS
      FROM DB2ADMIN.AST3 AS Q1
      WHERE (Q1.ROOT_LEVEL = 'Chemicals and Drugs')) AS Q2
   GROUP BY Q2.$C0) AS Q3
```

Query 3:

Note the following generalization of local predicates for this query:

- The COUNT(DISTINCT FACT1_SUBSET.DOC_ID) select list item is replaced by a column DOC_ID, and an addition of column DOC_ID to the GROUP BY list.
- There is no local predicate.

We see that a materialized view from the above generalization is a coarser granular version of AST3, since it only groups on MESH.ROOT_LEVEL. We can therefore potentially reuse AST3.

We ran an EXPLAIN of Query 2 to confirm that it was being routed to AST3 as shown in Example 2-41.

Example 2-41 EXPLAIN of Query 3

```
******************** EXPLAIN INSTANCE ********************

DB2_VERSION: 07.02.0
SOURCE_NAME: SQLC2D01
SOURCE_SCHEMA: NULLID
EXPLAIN_TIME: 2002-08-26-17.07.12.593002
EXPLAIN_REQUESTER: DB2ADMIN

Database Context:
----------------
    Parallelism: None
    CPU Speed: 7.478784e-007
    Comm Speed: 0
    Buffer Pool size: 5256
    Sort Heap size: 20000
    Database Heap size: 600
    Lock List size: 50
    Maximum Lock List: 22
    Average Applications: 1
    Locks Available: 1243

Package Context:
---------------
    SQL Type: Dynamic
    Optimization Level: 5
    Blocking: Block All Cursors
    Isolation Level: Cursor Stability



---------------- STATEMENT 1  SECTION 203 ----------------
    QUERYNO: 1
    QUERYTAG:
    Statement Type: Select
    Updatable: No
    Deletable: No
    Query Degree: 1

Original Statement:
------------------
SELECT MESH.ROOT_LEVEL, COUNT(DISTINCT FACT1_SUBSET.DOC_ID) COUNTS
FROM FACT1_SUBSET, MESH
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID
GROUP BY MESH.ROOT_LEVEL


Optimized Statement:
```

```
------------------
SELECT Q3.$C1 AS "ROOT_LEVEL", Q3.$C0 AS "COUNTS"
FROM
   (SELECT COUNT(Q2.$C1), Q2.$C0
   FROM
      (SELECT Q1.ROOT_LEVEL, Q1.COUNTS
      FROM DB2ADMIN.AST3 AS Q1) AS Q2
   GROUP BY Q2.$C0) AS Q3
```

Query 4:

The materialized view for this query looks like AST5 in Example 2-42. Note the following generalization of local predicates for this query:

- The predicates on columns MESH.ROOT_LEVEL and MESH.FIRST_LEVEL are added to the GROUP BY list, and removed from the predicates.
- The COUNT(DISTINCT FACT1_SUBSET.DOC_ID) select list item is replaced by a column DOC_ID, and an addition of column DOC_ID to the GROUP BY list.

Example 2-42 Materialized view AST5

```
CREATE SUMMARY TABLE AST5 AS
   (
   SELECT  ROOT_LEVEL, MESH.FIRST_LEVEL, AUTHOR_NAME, DOC_ID
   FROM  FACT1_SUBSET,  MESH, AUTHOR
   WHERE  FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
   FACT1_SUBSET.AUTHOR_ID=AUTHOR.AUTHOR_ID
   GROUP BY  FIRST_LEVEL, ROOT_LEVEL, AUTHOR_NAME, DOC_ID
   )
DATA INITIALLY DEFERRED REFRESH DEFERRED  IN USERSPACE1
```

An estimate of the size of the materialized view was well within the limit of an order of magnitude as compared to the base table. We ran an EXPLAIN of Query 4 to confirm that it was being routed to AST3 as shown in Example 2-43.

Example 2-43 EXPLAIN of Query 4

```
******************** EXPLAIN INSTANCE ********************

DB2_VERSION: 07.02.0
SOURCE_NAME: SQLC2D01
SOURCE_SCHEMA: NULLID
EXPLAIN_TIME: 2002-08-27-15.50.44.109000
EXPLAIN_REQUESTER: DB2ADMIN

Database Context:
----------------
```

```
   Parallelism: None
   CPU Speed: 7.478784e-007
   Comm Speed: 0
   Buffer Pool size: 5256
   Sort Heap size: 20000
   Database Heap size: 600
   Lock List size: 50
   Maximum Lock List: 22
   Average Applications: 1
   Locks Available: 1243

Package Context:
---------------
   SQL Type: Dynamic
   Optimization Level: 5
   Blocking: Block All Cursors
   Isolation Level: Cursor Stability



---------------- STATEMENT 1  SECTION 203 ----------------
   QUERYNO: 1
   QUERYTAG:
   Statement Type: Select
   Updatable: No
   Deletable: No
   Query Degree: 1

Original Statement:
------------------
SELECT AUTHOR.AUTHOR_NAME, COUNT(DISTINCT FACT1_SUBSET.DOC_ID) COUNTS
FROM FACT1_SUBSET, MESH, AUTHOR
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
        FACT1_SUBSET.AUTHOR_ID=AUTHOR.AUTHOR_ID AND MESH.ROOT_LEVEL='Anatomy'
        AND MESH.FIRST_LEVEL='Animal Structures'
GROUP BY AUTHOR.AUTHOR_NAME


Optimized Statement:
-------------------
SELECT Q3.$C1 AS "AUTHOR_NAME", Q3.$C0 AS "COUNTS"
FROM
   (SELECT COUNT(Q2.$C1), Q2.$C0
   FROM
      (SELECT Q1.AUTHOR_NAME, Q1.DOC_ID
      FROM DB2ADMIN.AST5 AS Q1
      WHERE (Q1.FIRST_LEVEL = 'Animal Structures') AND (Q1.ROOT_LEVEL =
           'Anatomy')) AS Q2
```

```
   GROUP BY Q2.$C0) AS Q3
```

Query 5:

Note the following generalization of local predicates for this query:

- The predicates on columns MESH.ROOT_LEVEL and MESH.FIRST_LEVEL are added to the GROUP BY list, and removed from the predicates. Note that the MESH.FIRST_LEVEL is an IN predicate.
- The COUNT(DISTINCT FACT1_SUBSET.DOC_ID) select list item is replaced by a column DOC_ID, and an addition of column DOC_ID to the GROUP BY list.

We see that a materialized view from the above generalization is a coarser granular version of AST5, since it only groups on MESH.ROOT_LEVEL and MESH.FIRST_LEVEL. We can therefore potentially reuse AST5. We ran an EXPLAIN of Query 4 to confirm that it was being routed to AST5 as shown in Example 2-44.

Example 2-44 EXPLAIN of Query 5

```
******************** EXPLAIN INSTANCE ********************

DB2_VERSION: 07.02.0
SOURCE_NAME: SQLC2D01
SOURCE_SCHEMA: NULLID
EXPLAIN_TIME: 2002-08-27-15.58.30.937000
EXPLAIN_REQUESTER: DB2ADMIN

Database Context:
----------------
   Parallelism: None
   CPU Speed: 7.478784e-007
   Comm Speed: 0
   Buffer Pool size: 5256
   Sort Heap size: 20000
   Database Heap size: 600
   Lock List size: 50
   Maximum Lock List: 22
   Average Applications: 1
   Locks Available: 1243

Package Context:
---------------
   SQL Type: Dynamic
   Optimization Level: 5
   Blocking: Block All Cursors
   Isolation Level: Cursor Stability
```

```
---------------- STATEMENT 1  SECTION 203 ----------------
   QUERYNO: 1
   QUERYTAG:
   Statement Type: Select
   Updatable: No
   Deletable: No
   Query Degree: 1

Original Statement:
------------------
SELECT AUTHOR.AUTHOR_NAME, COUNT(DISTINCT FACT1_SUBSET.DOC_ID) COUNTS
FROM FACT1_SUBSET, MESH, AUTHOR
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
        FACT1_SUBSET.AUTHOR_ID=AUTHOR.AUTHOR_ID AND MESH.ROOT_LEVEL='Anatomy'
        AND MESH.FIRST_LEVEL IN ('Body Regions','Cells')
GROUP BY AUTHOR.AUTHOR_NAME


Optimized Statement:
-------------------
SELECT Q3.$C1 AS "AUTHOR_NAME", Q3.$C0 AS "COUNTS"
FROM
   (SELECT COUNT(Q2.$C1), Q2.$C0
   FROM
      (SELECT Q1.AUTHOR_NAME, Q1.DOC_ID
      FROM DB2ADMIN.AST5 AS Q1
      WHERE (Q1.ROOT_LEVEL = 'Anatomy') AND Q1.FIRST_LEVEL IN ('Body Regions',
           'Cells')) AS Q2
   GROUP BY Q2.$C0) AS Q3
```

Query 6:

The materialized view for this query looks like AST6 in Example 2-45. Note the following generalization of local predicates for this query:

- The YEAR, MONTH, ROOT_LEVEL, FIRST_LEVEL, and COUNTRY_NAME columns in the GROUP BY list are all nullable, and therefore require the GROUPING function to be defined in the select list of the materialized view.

> **Note:** As explained in topic 2 on page 46, the GROUPING function must be defined for all nullable columns when super aggregates (ROLLUP, CUBE and grouping sets) are involved.

An estimate of the size of the materialized view was well within the limit of an order of magnitude as compared to the base table. After creating this materialized view, and populating it, we ran an EXPLAIN of Query 6 to confirm that it was being routed to AST6 as shown in Example 2-46.

Example 2-45 Materialized view AST6

```
CREATE SUMMARY TABLE AST6 AS
   (
   SELECT COUNTRY_NAME, COUNT(*) AS COUNT, YEAR, MONTH, ROOT_LEVEL,
   FIRST_LEVEL, GROUPING(COUNTRY_NAME) AS GPCNAME, GROUPING(YEAR) AS GPYEAR,
   GROUPING(MONTH) AS GPMONTH, GROUPING(ROOT_LEVEL) AS GPRLEVEL,
   GROUPING(FIRST_LEVEL) AS GPFLEVEL
   FROM FACT1_SUBSET, MESH, TIME, COUNTRY
   WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
   FACT1_SUBSET.TIME_ID=TIME.TIME_ID AND
   FACT1_SUBSET.COUNTRY_ID=COUNTRY.COUNTRY_ID
   GROUP BY ROLLUP(YEAR, MONTH), ROLLUP(ROOT_LEVEL, FIRST_LEVEL), COUNTRY_NAME
   )
DATA INITIALLY DEFERRED REFRESH DEFERRED  IN USERSPACE1
```

Example 2-46 EXPLAIN of Query 6

```
******************** EXPLAIN INSTANCE ********************

DB2_VERSION: 07.02.0
SOURCE_NAME: SQLC2D01
SOURCE_SCHEMA: NULLID
EXPLAIN_TIME: 2002-08-29-13.38.21.687000
EXPLAIN_REQUESTER: DB2ADMIN

Database Context:
----------------
   Parallelism: None
   CPU Speed: 7.478784e-007
   Comm Speed: 0
   Buffer Pool size: 5256
   Sort Heap size: 20000
   Database Heap size: 600
   Lock List size: 50
   Maximum Lock List: 22
   Average Applications: 1
   Locks Available: 1243

Package Context:
---------------
   SQL Type: Dynamic
   Optimization Level: 5
   Blocking: Block All Cursors
```

```
   Isolation Level: Cursor Stability


---------------- STATEMENT 1  SECTION 203 ----------------
   QUERYNO: 1
   QUERYTAG:
   Statement Type: Select
   Updatable: No
   Deletable: No
   Query Degree: 1

Original Statement:
------------------
with dt as
   (select cast(country_name as varchar(20)) as country_name, COUNT(*) as
           count, year, month, cast(root_level as varchar(20)) as root_level,
           cast(first_level as varchar(20)) as first_level
   from fact1_subset, mesh, time, country
   where fact1_subset.mesh_id=mesh.mesh_id and
           fact1_subset.time_id=time.time_id and
           fact1_subset.country_id=country.country_id
   group by rollup(year, month), rollup(root_level, first_level),
        country_name) select *
from dt
order by country_name, year, month, root_level, first_level

Optimized Statement:
-------------------
SELECT Q1.COUNTRY_NAME AS "COUNTRY_NAME", Q1.COUNT AS "COUNT", Q1.YEAR AS
        "YEAR", Q1.MONTH AS "MONTH", Q1.ROOT_LEVEL AS "ROOT_LEVEL",
        Q1.FIRST_LEVEL AS "FIRST_LEVEL"
FROM DB2ADMIN.AST6 AS Q1
ORDER BY Q1.COUNTRY_NAME, Q1.YEAR, Q1.MONTH, Q1.ROOT_LEVEL, Q1.FIRST_LEVEL
```

Query 7:

Note the following generalization of local predicates for this query:

- The YEAR, ROOT_LEVEL, and COUNTRY_NAME columns in the GROUP BY list are all nullable, and therefore require the GROUPING function to be defined in the select list of the materialized view.

We see that a materialized view from the above generalization is a coarser granular version of AST6, since it only groups on three of the five columns defined in AST6. We can therefore potentially reuse AST6.

We ran an EXPLAIN of Query 7 to confirm that it was being routed to AST6 as shown in Example 2-47.

Example 2-47 EXPLAIN of Query 7

```
******************** EXPLAIN INSTANCE ********************

DB2_VERSION: 07.02.0
SOURCE_NAME: SQLC2D01
SOURCE_SCHEMA: NULLID
EXPLAIN_TIME: 2002-09-04-17.02.40.562000
EXPLAIN_REQUESTER: DB2ADMIN

Database Context:
----------------
   Parallelism: None
   CPU Speed: 7.478784e-007
   Comm Speed: 0
   Buffer Pool size: 5256
   Sort Heap size: 20000
   Database Heap size: 600
   Lock List size: 50
   Maximum Lock List: 22
   Average Applications: 1
   Locks Available: 1243

Package Context:
---------------
   SQL Type: Dynamic
   Optimization Level: 5
   Blocking: Block All Cursors
   Isolation Level: Cursor Stability



---------------- STATEMENT 1  SECTION 203 ----------------
   QUERYNO: 1
   QUERYTAG:
   Statement Type: Select
   Updatable: No
   Deletable: No
   Query Degree: 1

Original Statement:
------------------
with dt as
   (select cast(country_name as varchar(20)) as country_name, COUNT(*) as
           count, year, cast(root_level as varchar(20)) as root_level
   from fact1_subset, mesh, time, country
```

```
   where fact1_subset.mesh_id=mesh.mesh_id and
           fact1_subset.time_id=time.time_id and
           fact1_subset.country_id=country.country_id
   group by rollup(year), rollup(root_level), country_name) select *
from dt
order by country_name, year, root_level


Optimized Statement:
-------------------
SELECT Q1.COUNTRY_NAME AS "COUNTRY_NAME", Q1.COUNT AS "COUNT", Q1.YEAR AS
        "YEAR", Q1.ROOT_LEVEL AS "ROOT_LEVEL"
FROM DB2ADMIN.AST6 AS Q1
WHERE (Q1.GPMONTH = 1) AND (Q1.GPFLEVEL = 1)
ORDER BY Q1.COUNTRY_NAME, Q1.YEAR, Q1.ROOT_LEVEL
```

Query 8:

The materialized view for this query looks like AST7 in Example 2-48. Note the following generalization of local predicates for this query:

- The predicates on columns MESH.ROOT_LEVEL and MESH.FIRST_LEVEL are added to the GROUP BY list, and removed from the predicates. Note that the MESH.FIRST_LEVEL is an IN predicate.
- The COUNT(*) function in the CASE expression in the select list is replaced by a COUNT(*), and the FACT1_SUBSET.DOC_ID column is added to the GROUP BY list. The CASE expression is removed from the select list altogether.

An estimate of the size of the materialized view was well within the limit of an order of magnitude as compared to the base table. After creating this materialized view, and populating it, we ran an EXPLAIN of Query 6 to confirm that it was being routed to AST7 as shown in Example 2-49.

Example 2-48 Materialized view AST7

```
CREATE SUMMARY TABLE AST7 AS
(
SELECT  ROOT_LEVEL, MESH.FIRST_LEVEL, AUTHOR_NAME, DOC_ID, COUNT(*) AS C
FROM  FACT1_SUBSET,  MESH, AUTHOR
WHERE  FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
FACT1_SUBSET.AUTHOR_ID=AUTHOR.AUTHOR_ID
GROUP BY  FIRST_LEVEL, ROOT_LEVEL, AUTHOR_NAME, DOC_ID
)
DATA INITIALLY DEFERRED REFRESH DEFERRED  IN USERSPACE1
```

Example 2-49 EXPLAIN of Query 8

```
******************** EXPLAIN INSTANCE ********************

DB2_VERSION: 07.02.0
SOURCE_NAME: SQLC2D01
SOURCE_SCHEMA: NULLID
EXPLAIN_TIME: 2002-08-29-13.27.27.781000
EXPLAIN_REQUESTER: DB2ADMIN

Database Context:
----------------
   Parallelism: None
   CPU Speed: 7.478784e-007
   Comm Speed: 0
   Buffer Pool size: 5256
   Sort Heap size: 20000
   Database Heap size: 600
   Lock List size: 50
   Maximum Lock List: 22
   Average Applications: 1
   Locks Available: 1243

Package Context:
---------------
   SQL Type: Dynamic
   Optimization Level: 5
   Blocking: Block All Cursors
   Isolation Level: Cursor Stability



---------------- STATEMENT 1  SECTION 203 ----------------
   QUERYNO: 1
   QUERYTAG:
   Statement Type: Select
   Updatable: No
   Deletable: No
   Query Degree: 1

Original Statement:
------------------
SELECT AUTHOR.AUTHOR_NAME, doc_id,
case
when fact1_subset.doc_id=1000
THEN count(*)
ELSE NULL END
FROM FACT1_SUBSET, MESH, AUTHOR
WHERE FACT1_SUBSET.MESH_ID=MESH.MESH_ID AND
```

```
        FACT1_SUBSET.AUTHOR_ID=AUTHOR.AUTHOR_ID AND MESH.ROOT_LEVEL='Anatomy'
        AND MESH.FIRST_LEVEL IN ('Body Regions','Cells')
GROUP BY AUTHOR.AUTHOR_NAME, doc_id


Optimized Statement:
-------------------
SELECT Q3.$C2 AS "AUTHOR_NAME", Q3.$C1 AS "DOC_ID",
CASE
WHEN (Q3.$C1 = 1000)
THEN Q3.$C0
ELSE NULL END
FROM
   (SELECT SUM(Q2.$C2), Q2.$C0, Q2.$C1
   FROM
      (SELECT Q1.DOC_ID, Q1.AUTHOR_NAME, Q1.C
      FROM DB2ADMIN.AST7 AS Q1
      WHERE (Q1.ROOT_LEVEL = 'Anatomy') AND Q1.FIRST_LEVEL IN ('Body Regions',
           'Cells')) AS Q2
   GROUP BY Q2.$C1, Q2.$C0) AS Q3
```

Attention: We did not conduct any performance measurements to determine the performance gains of routing to the materialized views. In a real world environment, you would need to continuously evaluate the efficacy of materialized views, and create and drop them as needed based on performance needs.

2.9 Materialized view tuning considerations

Two broad categories of tuning considerations apply to materialized views as follows:

- **User query related:** These are the considerations related to improving the performance of user queries against base tables that get routed to the materialized view. This includes ensuring that RUNSTATS is current, and that appropriate indexes exist on the materialized view.
- **Materialized view maintenance related:** These are considerations related to improving the performance of materialized view maintenance by DB2 when updates occur on the underlying tables. We recommend that you follow these guidelines:
  - Create a non-unique index on the materialized view columns that guarantee uniqueness of rows in a materialized view. Refer to “No duplicate rows in materialized view restriction” on page 94 for guidelines

on identifying these columns. Example 2-50 shows examples of columns that form unique keys in different materialized views.

Note: Unique indexes can *not* be defined on a materialized view.

In the case of a partitioned materialized view, the partitioning key should be a subset of the columns described above.

– Do *not* create an index on the staging table, since such indexes will degrade the performance of appends to the staging table.

– Create an informational or system enforced referential integrity (RI) constraint on joins in a materialized view if appropriate, since DB2 takes advantage of these constraints to optimize the maintenance of materialized views.

Consider a materialized view with a join between the primary key of the parent table and corresponding foreign key of the child table. DB2 takes advantage of such an RI constraint to eliminate maintenance operations on the materialized view. For example, DB2 deduces that an insert to the parent table will not affect the materialized view since the join is empty. That is, due to the RI constraint between the parent table and child table, an insert of a row in the parent table guarantees that there can be no matching rows in the child table.

It is more appropriate to create informational referential constraints to achieve this optimization, rather than system enforced referential constraints, since the latter has application development as well as operations impact. Example 2-25 on page 54 shows an example of system enforced and informational referential integrity constraints.

– Partition the staging table according to the partitioning of the materialized view to promote collocated joins.

In choosing indexes, you should also take into account any joins necessitated by REFRESH IMMEDIATE and staging materialized view maintenance operations that must be included in packages updating the base tables. An EXPLAIN of such packages will identify these maintenance operations which might benefit greatly from appropriate indexes on the joined columns.

Example 2-50 Columns that form unique keys in materialized views

```
-- Case 1: A materialized view with a simple GROUP BY
--
CREATE TABLE loc_status_summary(locid, status, total, count) AS
   (
   SELECT t.locid, t.status, sum(ti.amount), COUNT(*)
   FROM trans AS t, transitem AS ti
   WHERE t.transid = ti.transid
```

```
   GROUP BY t.locid, t.status
   )
   DATA INITIALLY DEFERRED REFRESH IMMEDIATE
```

-- The GROUP BY items form a unique key in the result of the materialized view.
*-- In the above case, columns **locid** and **status** form the unique key.*

```
-- Case 2: A materialized view with complex GROUP BY

CREATE TABLE loc_rollup(country, country_grouping, state,
      state_grouping, total, count) AS
   (
   SELECT l.country, grouping(l.country), l.state, grouping(l.state),
      sum(ti.amount), COUNT(*)
   FROM trans AS t, transitem AS ti, loc AS l
   WHERE t.transid = ti.transid and t.locid = l.locid
   GROUP BY ROLLUP(l.country, l.state)
   )
   DATA INITIALLY DEFERRED REFRESH IMMEDIATE
```

-- The GROUP BY items plus the GROUPING columns of the
-- nullable GROUP BY items form a unique key.
-- Columns loc.country and loc.state are nullable and so
-- GROUPING(l.country) and GROUPING(l.state) must be included
-- in the materialized view to avoid duplicate rows.
*-- The columns that form the unique key in the above example are **country,***
***-- country_grouping, state** and **state_grouping**.*

```
-- Case 3: A join materialized view without GROUP BY

CREATE TABLE trans_join(transid, acctid, transitemid, amount) AS
   (
   SELECT t.transid, t.acctid, ti.transitemid, ti.amount
   FROM trans AS t, transitem AS ti
   WHERE t.transid = ti.transid
   )
   DATA INITIALLY DEFERRED REFRESH IMMEDIATE
```

-- A unique key of each underlying table must appear in the result of the
-- materialized view query and the unique key of the materialized view is
-- the combination of those unique keys.
*-- In the above example, columns **transid** and **transitemid** (the keys of trans and*
-- transitem respectively) form the unique key of the materialized view.

2.10 Refresh optimization

REFRESH TABLE operations can have the following negative impacts:

- Refresh takes a z-lock on the materialized view thus making it unavailable for access by SQL queries. Performance can be significantly impacted for queries depending on materialized view optimization during the refresh window.
- Refresh also takes a z-lock on the staging table (if one exists). This can have a negative impact on updates to the base tables (they will not succeed), if refresh takes an extended period of time, since the staging table is updated in the same unit-of-work as updates to the base table.
- Refresh causes logging to occur as a consequence of updates to the materialized view, as well as pruning of the staging table. Refresh also consumes CPU, IO and buffer pool resources that impacts other users contending for the same resources.

Refresh resource consumption can be reduced by combining multiple materialized view refreshes in a single REFRESH TABLE statement. DB2 uses “multi-query optimization” to share joins and aggregations required of each materialized view in order to reduce the resource consumption against base tables shared by the materialized views. Figure 2-14 describes this process.

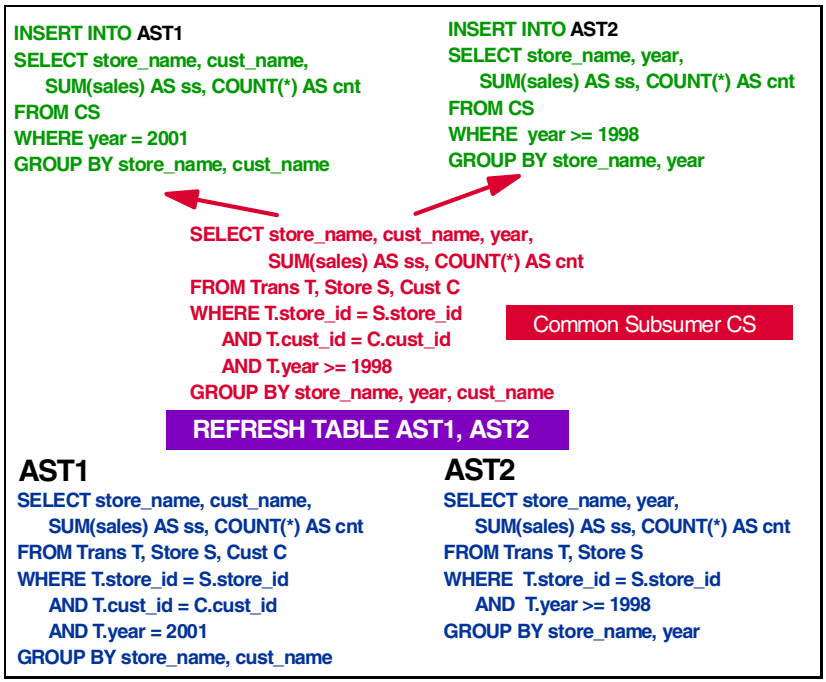


Figure 2-14 Multi-query optimization in REFRESH TABLE with materialized views

AST1 is a materialized view based on tables TRANS, STORE and CUST, while AST2 is based on tables TRANS and STORE.

Consider issuing the following:

```
REFRESH TABLE AST1, AST2
```

This causes DB2 to attempt to match the materialized view queries to formulate a “common subsumer” query CS, which is executed on the base tables, the results of which are then suitably predicated to update AST1 and AST2 respectively. This approach optimizes resource consumption against the base tables and staging tables. This has a positive impact on the performance of SQL queries, and updates of base tables associated with staging tables.

Considerations in grouping materialized views in a single REFRESH TABLE statement include:

- Identical or overlapping base tables.
- Identical latency requirements for both materialized views, or at least acceptable latency discrepancies between the materialized views.
- Large size of the base tables — significant performance gains can be achieved in such cases.

2.11 Materialized view limitations

The limitations applying to materialized views fall into the categories and subcategories shown in Figure 2-15.

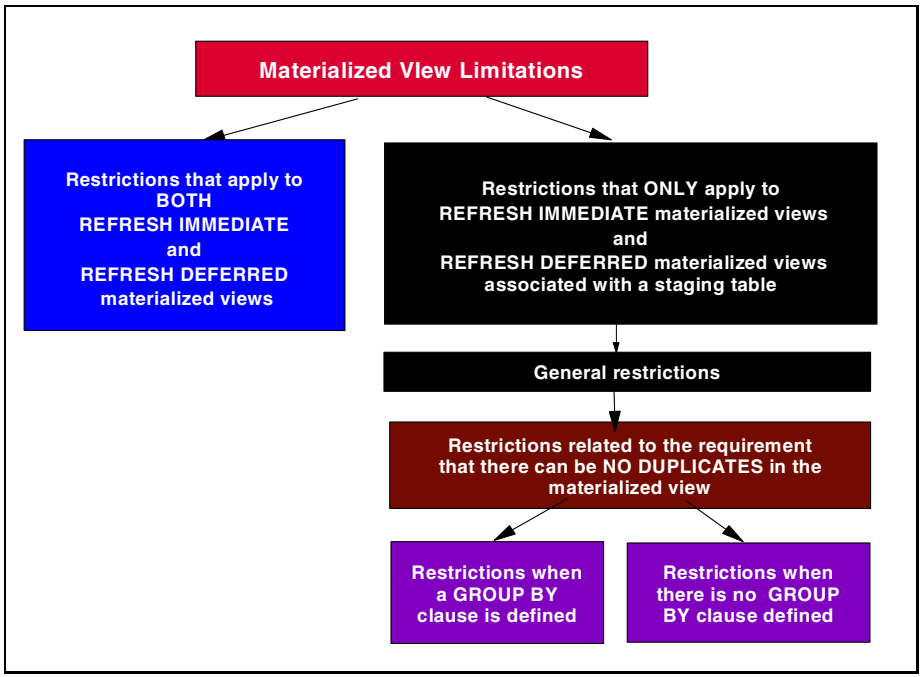


Figure 2-15 Materialized view limitation categories

The following sections describe the restrictions in each of the above categories.

2.11.1 REFRESH DEFERRED and REFRESH IMMEDIATE

The following fullselect restrictions apply to both REFRESH DEFERRED and REFRESH IMMEDIATE materialized views:

- References to a materialized view, declared temporary table, or typed table in any FROM clause.
- References to a view where the fullselect of the view violates any of the listed restrictions on the fullselect of a materialized view.
- Expressions that are a reference type or DATALINK type (or distinct type based on these types).
- Functions that have external action.
- Functions written in SQL.

- Functions that depend on physical characteristics (for example DBPARTITIONNUM, HASHEDVALUE).
- Table or view references to system objects (catalog tables). EXPLAIN tables should also not be specified.
- Expressions that are a structured type or LOB type (or a distinct type based on a LOB type).
- When REPLICATED is specified, the following restrictions apply:
  - GROUP BY clause is not allowed.
  - The materialized view query must only reference a single table, that is, not include a join.

2.11.2 REFRESH IMMEDIATE and queries with staging table

The following limitations *only* apply to REFRESH IMMEDIATE materialized views, and queries used to create REFRESH DEFERRED tables associated with a staging table.

General restrictions

- The fullselect must be a subselect, with the exception that UNION ALL is supported in the input table expression of a GROUP BY.
- The subselect can *not* include:
  - References to a nickname
  - Functions that are non-deterministic
  - Scalar fullselects
  - Predicates with fullselects
  - Special registers like CURRENT TIMESTAMP
  - SELECT DISTINCT
- If the FROM clause references more than one table or view, it can only define an inner join without using the explicit INNER JOIN syntax.
- GROUP BY restrictions

  When a GROUP BY clause is specified, the following considerations apply:

  - Column functions SUM, COUNT, COUNT_BIG and GROUPING (without DISTINCT) are supported. The select list must contain a COUNT(*) or COUNT_BIG(*) column. If the materialized view select list contains SUM(X) where X is a nullable argument, then the materialized view must also have COUNT(X) in its select list. These column functions can not be part of any expressions.
  - A HAVING clause is not allowed.

- If in a multiple partition database partition group, the partitioning key must be a subset of the GROUP BY items.

No duplicate rows in materialized view restriction

The materialized view must *not* contain duplicate rows, and the following restrictions specific to this uniqueness requirement apply, depending upon whether or not a GROUP BY clause is specified.

- When a GROUP BY is specified, the following uniqueness related restrictions apply:
  - All GROUP BY items must be included in the select list.
  - When the GROUP BY contains GROUPING SETS, CUBE or ROLLUP, then the GROUP BY items and associated GROUPING column functions in the select list must form a unique key of the result set. Thus, the following restrictions must be satisfied:
    - No grouping sets may be repeated. For example, ROLLUP(X,Y),X is not allowed because it is equivalent to GROUPING SETS((X,Y),(X),(X))
    - If X is a nullable GROUP BY item that appears within GROUPING SETS, CUBE, or ROLLUP, then GROUPING(X) must appear in the select list
- When a GROUP BY clause is *not* specified, the following uniqueness related restrictions apply:
  - The materialized view's non-duplicate requirement is achieved by deriving a unique key for the materialized view from one of the unique key constraints defined in each of the underlying tables. Therefore, the underlying tables must have at least one unique key constraint defined on them, and the columns of these keys must appear in the select list of the materialized view definition.

Note: DB2 sometimes allows you to define a REFRESH DEFERRED table even though it cannot use it for materialized view optimization. In such cases, it issues a warning SQL20059W (sqlstate 01633).

A typical scenario for this functionality is when a database administrator needs to:

1. Create a data mart based on detailed data from operational systems.
2. Provide direct access to end users to this data mart only, **without** allowing them access to the base detailed data.
3. Control the refresh cycle of this data mart via the REFRESH TABLE statement.

This is an example of creating an materialized view that is directly accessible by end users, and is not involved in materialized view optimization.

Certain operations cannot be performed on the base tables of a materialized view that needs to be incrementally maintained.

- IMPORT REPLACE cannot be used on an base table of a materialized view.
- ALTER TABLE NOT LOGGED INITIALLY WITH EMPTY TABLE cannot be done on a base table of a materialized view.
- Materialized views cannot be used as exception tables to collect information when constraints are being validated during bulk constraints checking (during LOAD or executing the SET INTEGRITY statement).

2.12 Replicated tables in nodegroups

In a partitioned database, the performance of join queries can be greatly enhanced through collocation of rows of the different tables involved in the join. Figure 2-16 describes such an environment, where the STORE and TRANS tables have been partitioned on *storeid* column. An SQL query that requires a join on the *storeid* column will see significant performance benefits from this partitioning scheme, because of the greater parallelism achieved through collocated joins.

However, when the CUST table is also involved in the join, then a collocated join is not possible, since the CUST table does not have a *storeid* column, and therefore cannot be partitioned by *storeid*. While DB2 UDB can choose to perform a directed join in this particular case[14], the performance of such joins is less efficient than that of a collocated join, since the movement of rows is inline with query execution.

Attention: We can now use materialized views to replicate tables to other nodes to enable collocated joins to occur even though the all the tables are not joined on the partitioned key. In Figure 2-16, CUST is replicated to the other nodes using the materialized view infrastructure in order to enable collocated joins for superior performance.

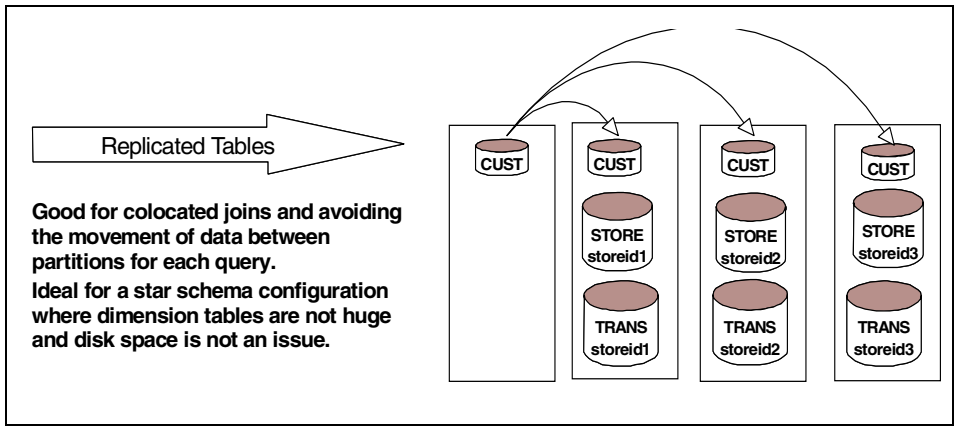


Figure 2-16 Collocation in partitioned database environment

Example 2-51 shows an example of creating such a materialized view, which creates an intra-database replica.

Example 2-51 Creating a replicated table in a nodegroup

```
CREATE TABLE custr1 AS
   (SELECT * FROM cust)
      DATA INITIALLY DEFERRED REFRESH IMMEDIATE REPLICATED IN <a nodegroup>
```

Attention: This form of replication should not be confused with the replication provided by DB2 DataPropagator, which is typically used for *inter-database* replication, that is, for replicating data between different databases. Table 2-2 summarizes the differences between materialized view intra-database replication, and DB2 DataPropagator inter-database replication.

[14] Broadcast or repartitioned joins can be used when the tables are not joined on the partitioning column.

Table 2-2 Intra-database replication versus inter-database replication

Replication type	Definition	Sources	Targets	Maintenance	Usage
Intra-database (materialized view)	CREATE TABLE statement	Local DB2 tables only	Local DB2 tables only	Synchronously maintained	Existence of materialized view is transparent to the application
Inter-database (DB2 Data Propagator)	Source and subscriptio ns defined through the DB2 Control Center, or the DataJoiner DJRA tool	Local or remote DB2 tables and views. Heterogeneous relational sources via DataJoiner nicknames. External data is staged in a consistent change data format.	Local or remote DB2 table. Heterogeneous relational targets supported via DataJoiner nicknames, with other external targets via consistent change data table interface.	On demand using subscription events and/or COPYONCE Apply startup option. ---------- Asynchronously updated from captured changes — either timer-driven or event-driven.	The application must explicitly reference the replica.

3

DB2 UDB's statistics, analytic, and OLAP functions

In this chapter we provide an overview of DB2 UDB's statistics, analytic, and OLAP functions.

3.1 DB2 UDB's statistics, analytic, and OLAP functions

DB2 UDB's query related functions are broadly classified into two categories:

- Statistics and analytic functions
- Online Analytical Processing (OLAP) functions

Appendix A, "Introduction to statistics and analytic concepts" on page 217 provides an introduction to some of the analytic concepts described here.

The following sections briefly describe these functions and provide examples to explain their usage where appropriate.

3.2 Statistics and analytic functions

The various analytic functions supported are listed in Table 3-1.

Table 3-1 List of statistics and analytic functions

Statistics and analytic functions	Description
AVG	Returns the average of a set of numbers.
CORRELATION or CORR	Returns the coefficient of correlation of a set of number pairs.
COUNT	Returns the count of the number of rows in a set of rows or values.
COUNT_BIG	Returns the number of rows or values in a set of rows or values. Result can be greater than the maximum value of integer.
COVARIANCE or COVAR	Returns the covariance of a set of number pairs.
MAX	Returns the maximum value in a set of values.
MIN	Returns the minimum value in a set of values.
RAND	Returns a random floating point number between 0 and 1
STDDEV	Returns the standard deviation of a set of numbers.
SUM	Returns the returns the sum of a set of numbers
VARIANCE or VAR	Returns the variance of a set of numbers.
Regression features:	
REGR_AVGX	Returns quantities used to compute regression diagnostic statistics
REGR_AVGY	Returns quantities used to compute regression diagnostic statistics.

Statistics and analytic functions	Description
REGR_COUNT	Returns the number of non-null number pairs used to fit the regression line.
REGR_INTERCEPT or REGR_ICPT	Returns the y-intercept of the regression line.
REGR_R2	Returns the coefficient of determination for the regression.
REGR_SLOPE	Returns the slope of the regression line.
REGR_SXX	Returns quantities used to compute regression diagnostic statistics.
REGR_SXY	Returns quantities used to compute regression diagnostic statistics.
REGR_SYY	Returns quantities used to compute regression diagnostic statistics.

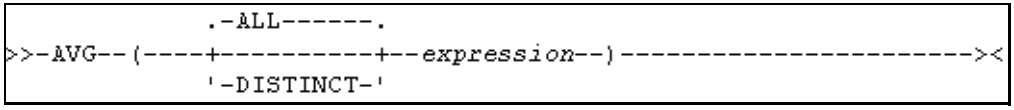
3.2.1 AVG

The AVG function returns the average of a set of numbers.

ALL indicates duplicate rows are to be included, and this is the default.

The average function is applied to a set of values after eliminating all null values. If DISTINCT is specified, duplicate values are eliminated as well.

Note: ALL and DISTINCT have the same meaning in other functions where they are supported.



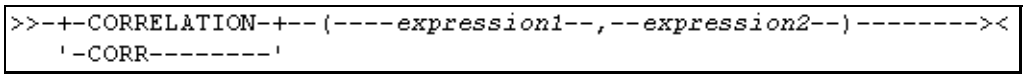
3.2.2 CORRELATION

The CORRELATION function returns the coefficient of correlation of a set of number pairs. The coefficient indicates the strength of the linear relationship between the set of variables.

The input values must be numeric, and the data type of the result is double-precision floating point.

The function is applied to the set of numeric pairs derived from the argument values (*expression 1,expression2*) by the elimination of all pairs for which either *expression1* or *expression2* is null.

- A null result implies the input set is empty.
- When the result is not null, it will be between minus one and one.
- A zero value means the two expressions are not linearly related.
- A minus one or a plus one mean they are linearly perfectly related.

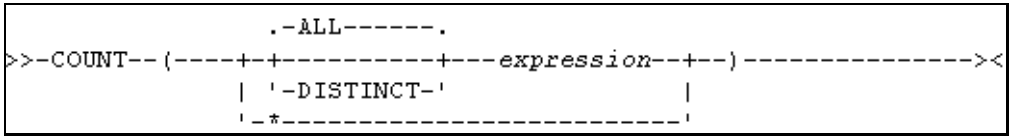


An example of correlation is shown in, “CORRELATION examples” on page 112.

3.2.3 COUNT

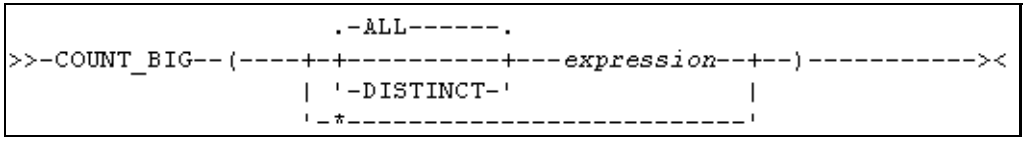
The COUNT function counts the number of rows or values in a set of rows or values. A row that includes only NULL values is included in the count, thus the result cannot be null. The result is a large integer.

The result is the number of rows in the set.



3.2.4 COUNT_BIG

The COUNT_BIG function counts the number of rows or values in a set of rows or values. It functions the same as COUNT except that the result can be greater than the maximum value of integer. The result data type of COUNT_BIG is a decimal with precision 31 and scale 0. Nulls are treated like they are in COUNT.



3.2.5 COVARIANCE

Covariance is a measure of the linear association between two variables. It is typically used as an intermediate computation enroute to other statistics functions such as the correlation coefficient.

The covariance value depends upon the units in which each variable is measured, unlike the case of the correlation coefficient.

The COVARIANCE function calculates the population[1] covariance of a set of number pairs. If both variables tend to be above or below the average simultaneously, then the covariance is positive. If one variable tends to have above-average values when the other variable has below average values, then the covariance is negative.

Input to the COVARIANCE function is a set of numeric pairs, and the output is a double-precision floating point type.

The function is applied to the set of numeric pairs derived from the argument values (*expression 1,expression2*) by the eliminating all pairs wherein either *expression1* or *expression2* is null.

A null result indicates an empty input set.

```
>>-+-COVARIANCE-+--(----expression1--,--expression2--)---------><
   '-COVAR------'
```

An example of covariance is shown in “COVARIANCE example” on page 111.

3.2.6 MAX

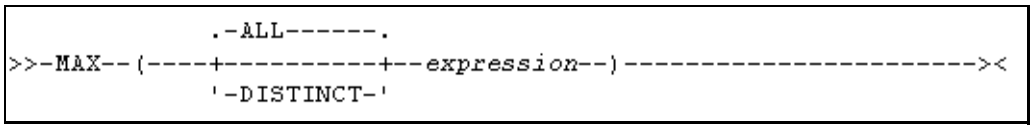
The MAX function returns the maximum value in a set of values.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is a null value.

Note: The specification of DISTINCT has no effect on the result, and is therefore not recommended. It is included for compatibility with other relational systems.

[1] A population is a collection of all data points for a given subject of interest.



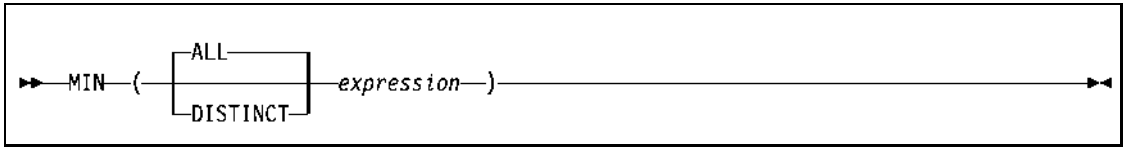
3.2.7 MIN

The MIN function returns the minimum value in a set of values.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is a null value.

Note: The specification of DISTINCT has no effect on the result, and is therefore not recommended. It is included for compatibility with other relational systems.

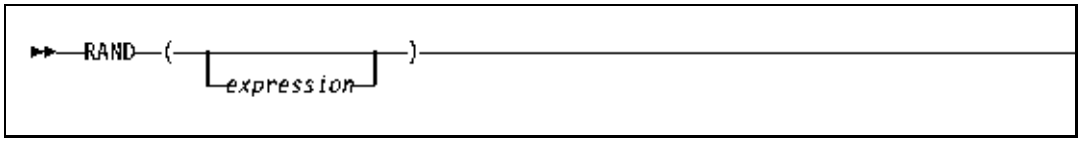


3.2.8 RAND

The RAND function returns a random floating point value between 0 and 1 using the argument as the optional seed value. The function is defined as non-deterministic.

An argument is not required, but if it is specified it can be either an INTEGER or SMALLINT. Providing a seed value guarantees the repeatability of the result, and is generally used for debugging purposes.

The result can be null, if the argument is null.



Executing the SQL in Example 3-1 results in a 10% sample (corresponding to 0.1) of all the rows in the CUSTOMERS table.

Example 3-1 RAND function

```
SELECT * FROM CUSTOMERS
WHERE RAND() < 0.1
```

Note that this is a "Bernoulli[2] sample". In the above SQL, if there were 100,000 rows in the CUSTOMERS table, the actual number of rows in the sample is random, but is equal on average to (100,000 / 10) = 10,000.

Since this technique involves a complete scan of the CUSTOMERS table, it is appropriate in situations where a sample is created once, and then used repeatedly in multiple queries. In other words, the cost of creating the sample is amortized over multiple queries.

3.2.9 STDDEV

The STDDEV function returns the population standard deviation (as opposed to the sample standard deviation) of a set of numbers.

The relationship between population standard deviation (SD_{pop}) and sample standard deviation (SD_{samp}) is as follows:

$$SD_{pop} = \sqrt{\frac{(n-1)}{n}} \times SD_{samp}$$

Where:

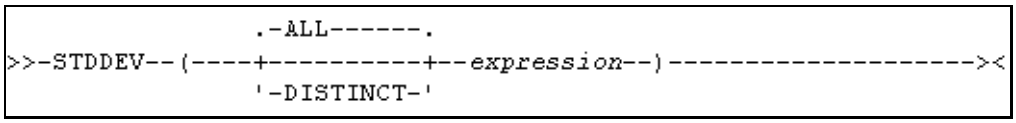
n is the population size.

The input must be numeric and the output is double-precision floating point.

The STDEV function is applied to the set of values derived from the argument values by the elimination of null values.

If the input data set is empty the result is null. Otherwise, the result is the standard deviation of the values in the set.

[2] In Bernoulli sampling, each row is selected for inclusion in the sample with probability q=(n/N) where ‘n’ is the desired sample size, and ‘N’ is the total number of rows and rejected with probability (1-q), independently of the other rows. The final sample size is random, but is equal to ‘n’ on average.

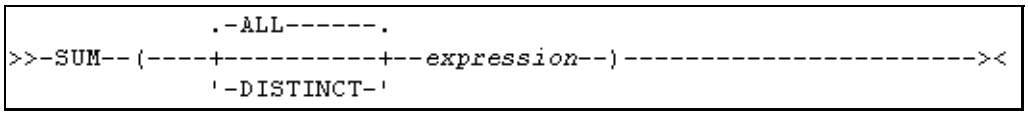


An example of standard deviation is shown in “STDDEV examples” on page 113.

3.2.10 SUM

The SUM function returns the sum of a set of numbers. The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the sum of the values in the set.



3.2.11 VARIANCE

The VARIANCE function returns the population variance (as opposed to the sample variance) of a set of numbers.

The relationship between population variance (Var_{pop}) and sample variance (Var_{samp}) is as follows:

$$Var_{pop} = \frac{(n-1)}{n} \times Var_{samp}$$

Where:

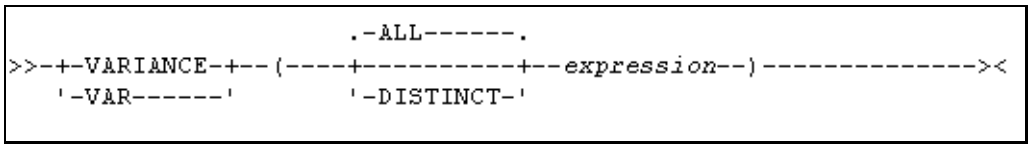
n is the population size.

The argument values must be numeric.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is a null value.

An example of variance is shown in, “VARIANCE example” on page 113.



3.2.12 Regression functions

The regression functions support the fitting of an ordinary-least-squares regression line of the form:

$$Y = aX + b$$

Where:

Y is the dependent variable.

X is the independent variable.

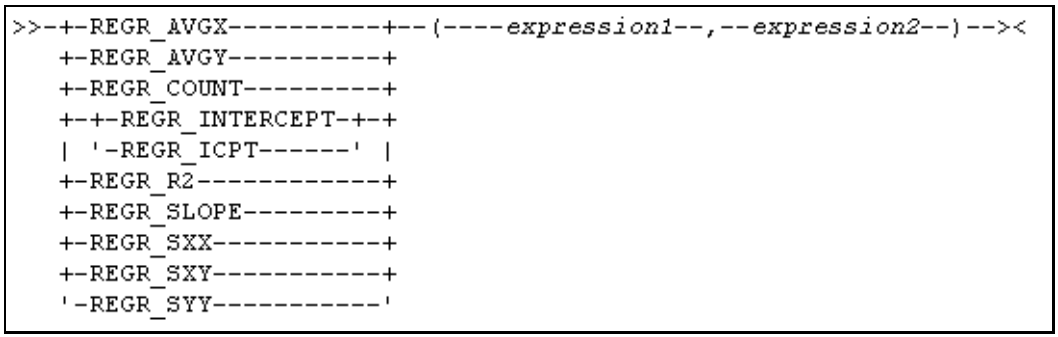
a is the slope of the regression line.

b is the y-intercept.

Both ‘a’ and ‘b’ are called coefficients.

There are nine distinct regression functions. They are:

REGR_SLOPE	Calculates the slope of the line (the parameter ’a’ in the above equation).
REGR_INTERCEPT	(REGR_ICPT) calculates the y-intercept of the regression line (’b’ in the above equation).
REGR_COUNT	Determines the number of non-null pairs used to determine the regression.
REGR_R2	Expresses the quality of the best-fit regression. (R-squared) is referred to as the coefficient of determination or the ’goodness-of-fit’ for the regression.
REGR_AVGX	Returns quantities that can be used to compute various diagnostic statistics needed for the evaluation of the quality and statistical validity of the regression model (They are defined further in this section).
REGR_AVGY	(Refer to the foregoing description.)
REGR_SXX	(Refer to the foregoing description.)
REGR_SYY	(Refer to the foregoing description.)
REGR_SXY	(Refer to the foregoing description.)



Important: Each function is applied to the set of values derived from the input numeric pairs (*expression1*,*expression2*) by the **elimination of all pairs for which either** *expression1* **or** *expression2* **is null.** In other words, both values must be non-null to be considered for the function.

Attention: *expression1* corresponds to the Y variable and *expression2* corresponds to the X variable.

The following considerations apply to regression functions:

- The input for all of the regression functions must be numeric.
- The output of REGR_COUNT is integer and all the remaining functions output in double-precision floating point.
- The regression functions are all computed simultaneously during a single pass through the data set.
- If the input set is not empty, and after elimination of the null pairs:
  - VARIANCE(*expression2*) is positive, then REGR_COUNT returns the number of non-null pairs in the set, and the remaining functions return results that are defined in Table 3-2.
  - VARIANCE(expression2) is equal to zero, then the regression line either has infinite slope or is undefined. In this case, the functions REGR_SLOPE, REGR_INTERCEPT, and REGR_R2 each return a null value, and the remaining functions return values defined in Table 3-2.
- If the input set is empty, REGR_COUNT returns zero, and the remaining functions return a null value.

- When the result is not null:
  - REGR_R2 is between 0 and 1.
  - REGR_SXX and REGR_SYY is non-negative. This non-negative value is used to describe the spread of the values for either X or Y from the their average values.

Table 3-2 Function computations

Function	Computation
REGR_SLOPE(*expr1,expr2*)	**COVAR**(*expr1,expr2*)/**VAR**(*expr2*)
REGR_ICPT(*expr1,expr2)*	**AVG**(*expr1*) - **REGR_SLOPE**(*expr1*, *expr2*) * **AVG**(*expr2*)
REGR_R2(*expr1*, *expr2*)	**POWER(CORR**(*expr1*, *expr2*), 2) if **VAR**(*expr1*)>0
REGR_R2(expr1,expr2)	**1** if **VAR**(expr1) = 0
REGR_AVGX(expr1,expr2)	**AVG**(expr2)
REGR_AVGY(*expr1*, *expr2*)	**AVG**(*expr1*)
REGR_SXX(*expr1*, *expr2*)	**REGR_COUNT**(*expr1*, *expr2*) * **VAR**(*expr2*)
REGR_SYY(*expr1*, *expr2*)	**REGR_COUNT**(*expr1*, *expr2*) * **VAR**(*expr1*)
REGR_SXY(*expr1*, *expr2*)	**REGR_COUNT**(*expr1*, *expr2*) * **COVAR**(*expr1*, *expr2*)

Important: The difference between REGR_AVG and AVG is that **all nulls** are **excluded** in the REGR_AVG computations, while they are **included** in the AVG(expression) computation.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

Tip: In general, it is more efficient even in the absence of null values, to use the regression functions to compute the statistics needed for a regression analysis, than to perform the equivalent computations using ordinary column functions such as AVG, VARIANCE, and COVARIANCE.

The usual diagnostic statistics that accompany a linear regression analysis can be computed in terms of the above functions as follows, and are offered with minimal explanation. All the following expressions apply to a simple linear regression, that is a model which includes only one independent variable.

- Adjusted R^2

  ```
  1 - ((1-REGR_R2)*((REGR_COUNT - 1) / (REGR_COUNT - 2)))
  ```

- Standard error (standard deviation of the residuals):

  ```
  SQRT((REGR_SYY-(POWER(REGR_SXY,2)/REGR_SXX)) / (REGR_COUNT-2))
  ```

  Note that:

  $$i_{th}residual = y_i - (ax_i + b)$$

- Total sum of squares:

  ```
  REGR_SYY
  ```

- Regression sum of squares:

  ```
  POWER(REGR_SXY,2) / REGR_SXX
  ```

- Residual sum of squares:

  ```
  (Total sum of squares) - (regression sum of squares)
  ```

- t statistic:

  For each coefficient (slope and intercept in the simple linear regression model), there is a concern as to whether the coefficient's value is meaningful, or if the coefficient is really zero. That is, the independent variable (x) does not contribute to the value of the dependent variable (y). The 't statistic' can help make this determination

  t statistic for slope

  ```
  REGR_SLOPE * SQRT(REGR_SXX) / (Standard error)
  ```

  t statistic for intercept

  ```
  REGR_INTERCEPT/((Standard error) *
  SQRT((1/REGR_COUNT)+(POWER(REGR_AVGX,2)/REGR_SXX))
  ```

3.2.13 COVAR, CORR, VAR, STDDEV, and regression examples

The following examples give a flavor of the use of these functions in a number of scenarios.

COVARIANCE example

We wish to explore the relationship between employee salary and the bonus that they receive, using the data shown in Figure 3-1.

FIRSTNME	SALARY	BONUS
IRVING	32250.00	500.00
BRUCE	25280.00	500.00
ELIZABETH	22250.00	400.00
MASATOSHI	24680.00	500.00
MARILYN	21340.00	500.00
JAMES	20450.00	400.00
DAVID	27740.00	600.00
WILLIAM	18270.00	400.00
JENNIFER	29840.00	600.00

Figure 3-1 D11 Employee salary & bonus

The DB2 SQL for covariance could be as shown in Example 3-2:

Example 3-2 COVARIANCE example

```
SELECT COVARIANCE (salary,bonus)
FROM employee
WHERE workdept = 'D11'
```

The result of this query is 23650.86.

This positive result indicates there is a positive relationship between salary and bonus, that is, employees with high (low) salaries tend to get high (low) bonuses.

While this conclusion appears intuitive with only a few data points, it is less obvious when there are a large number of data points involved — say 1000 or 10,000 employees. The covariance function thus enables relationships between variables.

Note: Covariance by itself does not indicate how strong the relationship is. It merely indicates one exists and whether it is a positive or negative relation. To determine the strength of a relationship the correlation must be calculated.

Correlation helps quantify the strength of the relationship.

CORRELATION examples

Using the same salary bonus example in Figure 3-1, we can quantify the strength of the relationship with the SQL shown in Example 3-3:

Example 3-3 CORRELATION example 1

```
SELECT CORRELATION (salary,bonus) AS cor
FROM employee
WHERE workdept = 'D11'
```

The result of the query is 0.739.

This quantitatively confirms the reasonably strong linear relationship between salary and bonus from the employees in department 'D11'.

Another example of correlation involving the retail industry is shown in Example 3-4. Assume we have the transactions of purchases from all the customers of a a retail organization selling a variety of products, and we would like to identify customers with similar buying habits. For example, when Customer A bought a particular product, Customer B also tended to buy the same product. Such information can be put to effective use in targeted marketing.

A view called *transhist* is created that contains the customer id, product id, and the dollar amount purchased over all transactions.

Example 3-4 CORRELATION example 2

```
SELECT a.custid as custid1, b.custid as custid2,
   CORR(a.amount, b.amount) AS cor
FROM transhist a, transhist b
WHERE a.prodid = b.prodid AND a.custid < b.custid
GROUP BY a.custid, b.custid
HAVING CORR(a.amount, b.amount) >= 0.5 AND COUNT(*) > 100
ORDER BY a.custid, cor DESC
```

This query joins the view with itself, and uses the HAVING clause to restrict the output to cases of high correlation (>= 0.5), and to cases where there are at least a 100 products involved, that is, there are at least 100 data points used to compute the correlation.

The result of this query is as follows:

```
CUSTID1     CUSTID2     CORR
----------- ----------- -----
1026        8271         0.51
1071        2014         0.74  <=
1071        7219         0.63
2014        7219         0.58
8271        9604         0.56
```

The result shows a high correlation between the buying habits of Customer 1071 and Customer 2014, that is, whenever customer 1071 bought a large amount of a given product, then customer 2014 also tended to buy a large amount of the same product.

VARIANCE example

DB2 has a built-in function to calculate variance. Using the same salary and bonus data shown in Figure 3-1, our SQL is shown in Example 3-5:

Example 3-5 VARIANCE example

```
SELECT AVG(salary), VARIANCE(salary) AS Variance
FROM  employee
WHERE workdept = 'D11'
```

The average salary is $24677.78, while the variance in our case is 1.885506172839506E7.

However this is not very intuitive, and standard deviation provides a more intuitive answer.

STDDEV examples

Using the same data as shown in Figure 3-1, the standard deviation of salary of employees in department ‘D11’ an be computed as shown in Example 3-6:

Example 3-6 STDDEV example 1

```
SELECT AVG(salary), STDDEV(salary) AS StandDev
FROM  employee
WHERE workdept = 'D11'
```

The result of this query is an average of $2477.78 and a standard deviation of $4342.24.

It indicates the variation of individual salaries from the average salary for the set, and is more intuitive than the variance function discussed earlier.

Another example of standard deviation involves computing the various statistics of an organization’s sales worldwide over multiple years.

The data is contained in three tables: trans, transitem, and loc, as shown in Example 3-7.

Example 3-7 STDDEV example 2

```
SELECT loc.country AS country, YEAR(t.pdate) AS year,
   COUNT(*) AS count, SUM(ti.amount) AS sum,
   AVG(ti.amount) AS avg, MAX(ti.amount) AS max,
   STDDEV(ti.amount) AS std
FROM trans t, transitem ti, loc loc
WHERE t.transid = ti.transid AND loc.locid = t.locid
GROUP BY loc.country, year(t.pdate)
```

The result of this query is as follows:

```
country    year   count     sum     avg     max  stddev
-------  ------  ------  ------  ------  ------  ------
    USA    1998     235  127505  542.57  899.99   80.32
    USA    1999     349  236744  678.35  768.61  170.45
GERMANY    1998     180   86278  479.32  771.65   77.41
GERMANY    1999     239  126737  530.28  781.99   72.22
...
```

The result shows commonly gathered statistics related to sales such as COUNT, SUM, AVG and MAX. The STDDEV function shows that USA sales in 1999 are much more variable (STDDEV of $170.45) than sales in other years and other locations, i.e., the amounts in the individual sales transactions vary more widely from their average value of $678.35.

Linear Regression examples

Using the same data shown in Figure 3-1, we will derive a regression model where salary is the independent variable and bonus is the dependent variable using the DB2 SQL shown in Example 3-8.

Example 3-8 Linear regression example 1

```
SELECT REGR_SLOPE (bonus , salary) AS slope,
REGR_ICPT (bonus , salary) AS intercept
FROM employee
WHERE workdept = 'D11'
```

The result of this query is a slope of 0.0125 and an intercept is $179.313, that is:

$$Bonus = 0.0125 \times Salary + 179.313$$

Note: The columns referenced in the regression functions are reversed from those in the variance and covariance examples. Since we wish to determine BONUS as a function of SALARY, it is listed first before SALARY.

DB2 has a $\mathbf{R}^2$ function, REGR_R2. The properties of $\mathbf{R}^2$ are:

- $\mathbf{R}^2$ bound is between 0 and 1.
- If $\mathbf{R}^2$ equals 1 then all the points fit on the regression line exactly.
- If $\mathbf{R}^2$ equals zero then the two attributes are independent.

The closer $\mathbf{R}^2$ is to 1, the better the computed linear regression model. In general, an $\mathbf{R}^2$ greater than 0.75 or so, is considered a good fit for most applications. However, it varies by application and it is ultimately up to the user to decide what value constitutes a good model.

The DB2 SQL could look as shown in Example 3-9:

Example 3-9 Linear regression example 2

```
SELECT REGR_R2 (bonus , salary) AS r2
FROM employee
WHERE workdept = 'D11'
```

The result of this query is 0.54624.

Since $\mathbf{R}^2$ is not very close to 1, we conclude that the computed linear regression model does not appear to be a very good fit.

Another example of using regression involves the assumption of a linear relationship between the advertising budget and sales figures of a particular organization that conforms to the equation:

$$y = ax + b$$

Where:

'y' is the sales dependent variable.

'x' is the advertising budget independent variable.

'a' is the slope.

'b' is the y-axis intercept corresponding to budget cost even with zero sales.

The queries shown in Example 3-10 determine the values for ‘a’, and ‘b’ given a set of non-null values of budget and sales data points in a table ‘t’:

Example 3-10 Linear regression example 3

```
SELECT
  REGR_COUNT(t.sales, t.ad_budget) AS num_cities,
  REGR_SLOPE(t.sales, t.ad_budget) AS a,
  REGR_ICPT(t.sales, t.ad_budget) AS b
FROM t
```

The result of the query is as follows, with REGR_COUNT returning the number of (x,y) **non-null pairs** used to fit the regression line.

```
num_cities       a      b
----------  ------ ------
126         1.9533 13.381
```

The input data and the derived linear model are shown in Figure 3-2.

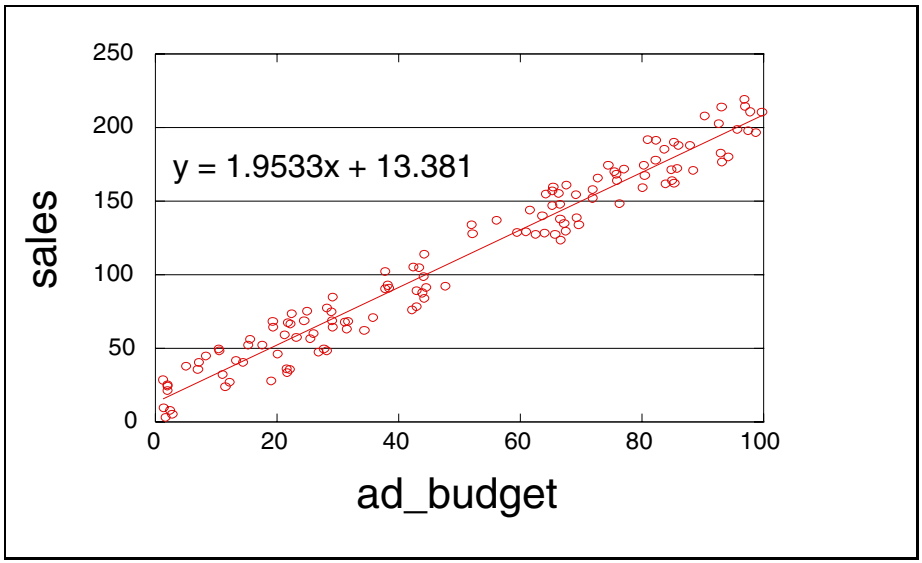


Figure 3-2 Linear regression

While the foregoing SQL models the equation, it does not tell you the quality of the fit, that is, the accuracy of the regression line. As described earlier, the $\mathbf{R}^2$ statistic must be computed to determine the quality of the regression line. $\mathbf{R}^2$ is the square of the correlation coefficient (CORR). $\mathbf{R}^2$ can also be interpreted as the proportion of variation in the ‘y’ values that is explained by the variation in the ‘x’ values, as opposed to variation due to randomness or to other variables not included in the model. Consider the coding shown in Example 3-11.

Example 3-11 Linear regression example 4

```
SELECT
  REGR_COUNT(t.sales, t.ad_budget) AS num_cities,
  REGR_SLOPE(t.sales, t.ad_budget) AS a,
  REGR_ICPT(t.sales, t.ad_budget) AS b,
  REGR_R2(t.sales, t.ad_budget) as rsquared
FROM t
```

The result of this query is as follows. It shows $\mathbf{R}^2$ to be 0.95917, which is a very high quality of fit of the regression line.

```
num_cities       a      b  rsquared
----------  ------ ------  ---------
128         1.9533 13.381    0.95917
```

Important: DB2 supports non-linear regression models involving a single independent variable. For example,

$$y = a x^2 + b$$

Restriction: We do *not* support regression models involving more than one independent variable. For example,

$$\mathbf{y = a_1x_1 + a_2x_2 + ...a_nx_n + b}$$

3.3 OLAP functions

OLAP is “a category of software technology hat enables analysts, managers and executives to gain insight into data through fast, consistent, interactive access to a wide variety of possible views of information that has been transformed from raw data to reflect the real dimensionality of the enterprise as understood by the user”.[3]

Typical enterprise dimensions are time, location/geography, product and customer.

While OLAP systems have the ability to answer “who” and “what” questions, it is their ability to answer “what if” and “why” that sets it apart from data warehouses. OLAP enables decision making about future actions.

[3] BI Certification Guide (SG24-5747)

OLAP functions provide the ability to return the following information in a query result:

- Ranking with RANK & DENSE_RANK.
- Numbering with ROW_NUMBER.
- Aggregation with existing column functions such as MAX, MIN, AVG etc.

Key to OLAP functions is the ability to define the set of rows over which the function is applied, and the sequence in which the function is applied. This set of rows is called a window. When an OLAP function is used with a column function, like AVG, SUM, MAX, etc., the target rows can be further refined, relative to the current row, as either a range, or a number of rows preceding and following the current row. For example, within a window partitioned by month, a moving average can be calculated over the previous three month period.

Besides windowing, the ability to group sets of rows is critical to OLAP functionality. ROLLUP and CUBE are extensions to the GROUP BY clause to provide OLAP functionality. ROLLUP and CUBE are called super-groups.

We discuss OLAP functionality as:

- Ranking, numbering and aggregate functions
- GROUPING capabilities ROLLUP & CUBE

We then follow it with examples.

3.3.1 Ranking, numbering and aggregation functions

Figure 3-3, Figure 3-4, and Figure 3-5 provide an overview of the syntax of some of the OLAP functions.

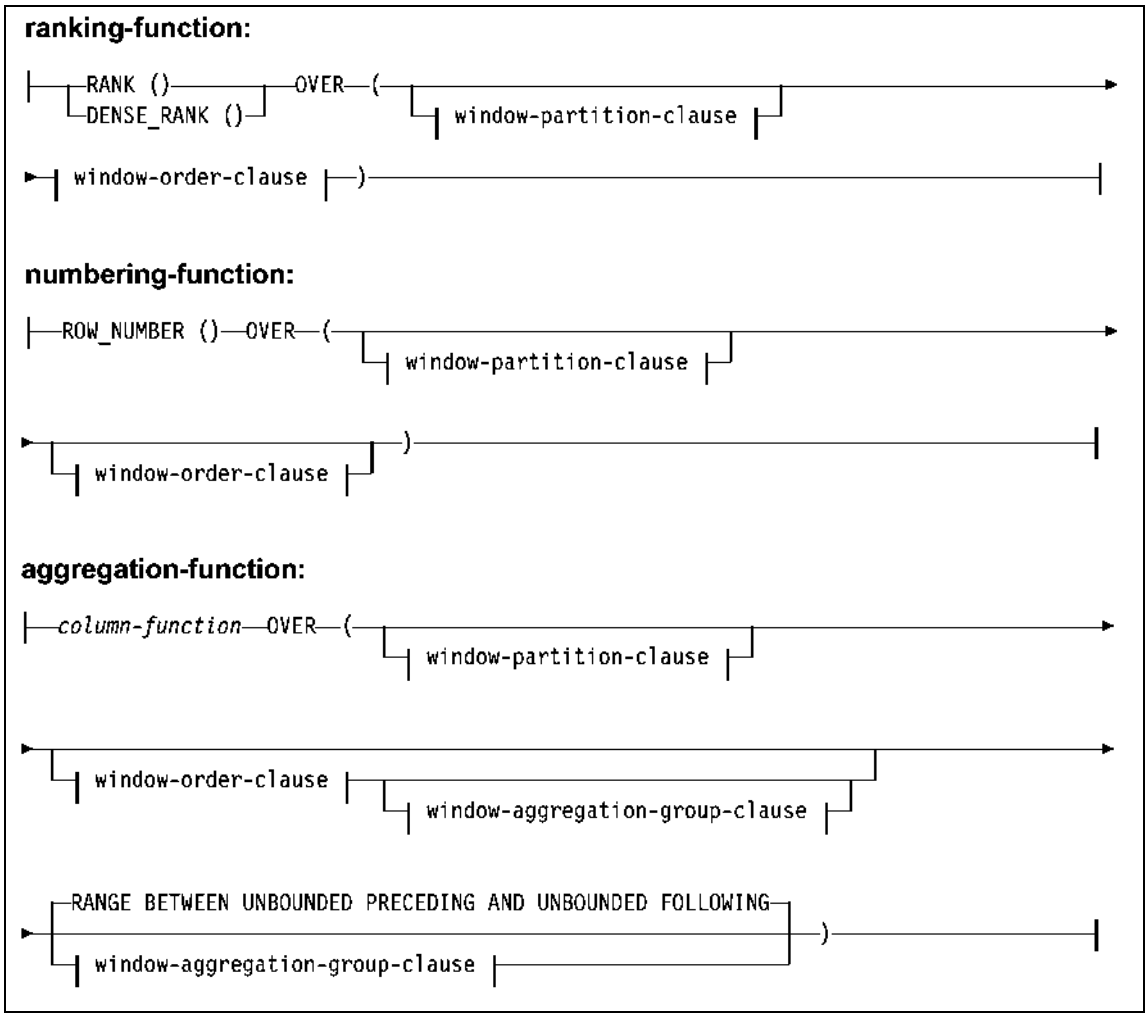


Figure 3-3 Ranking, numbering and aggregate functions

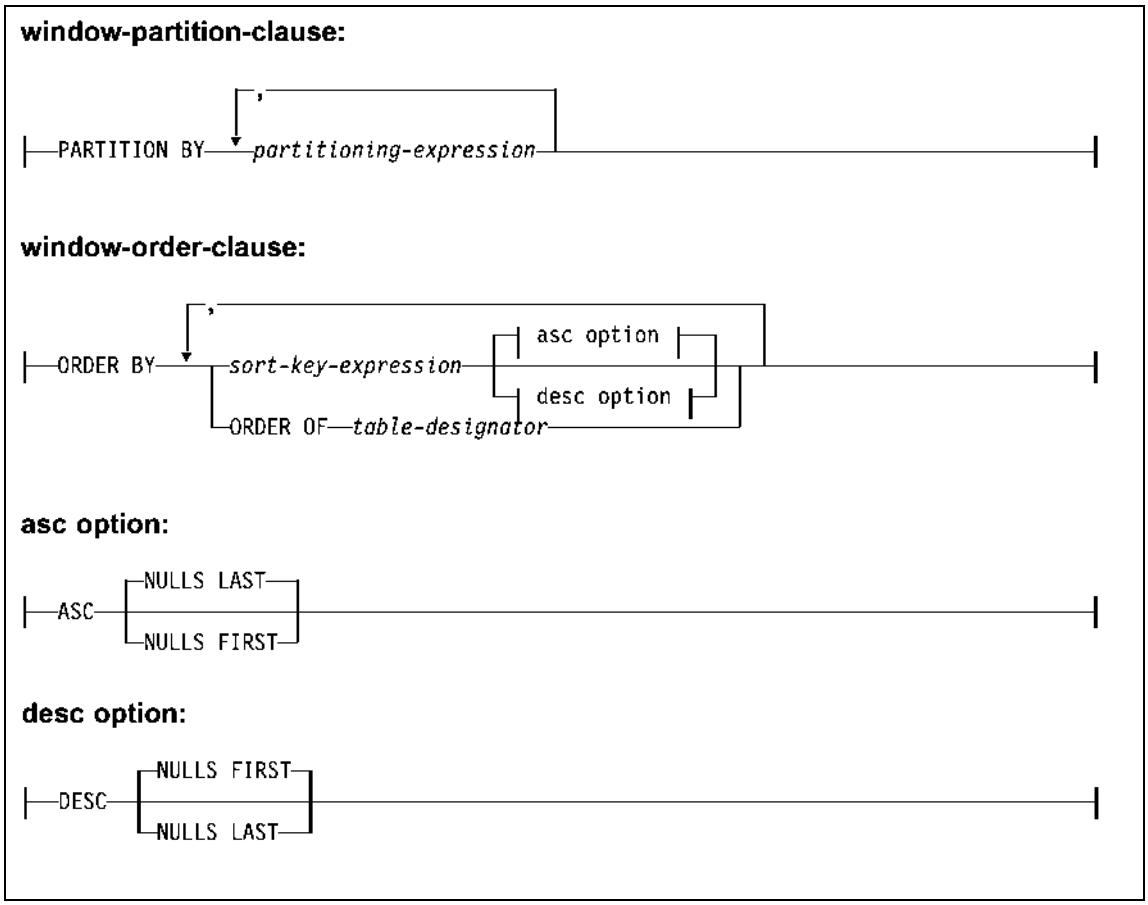


Figure 3-4 Window partition and window order clauses

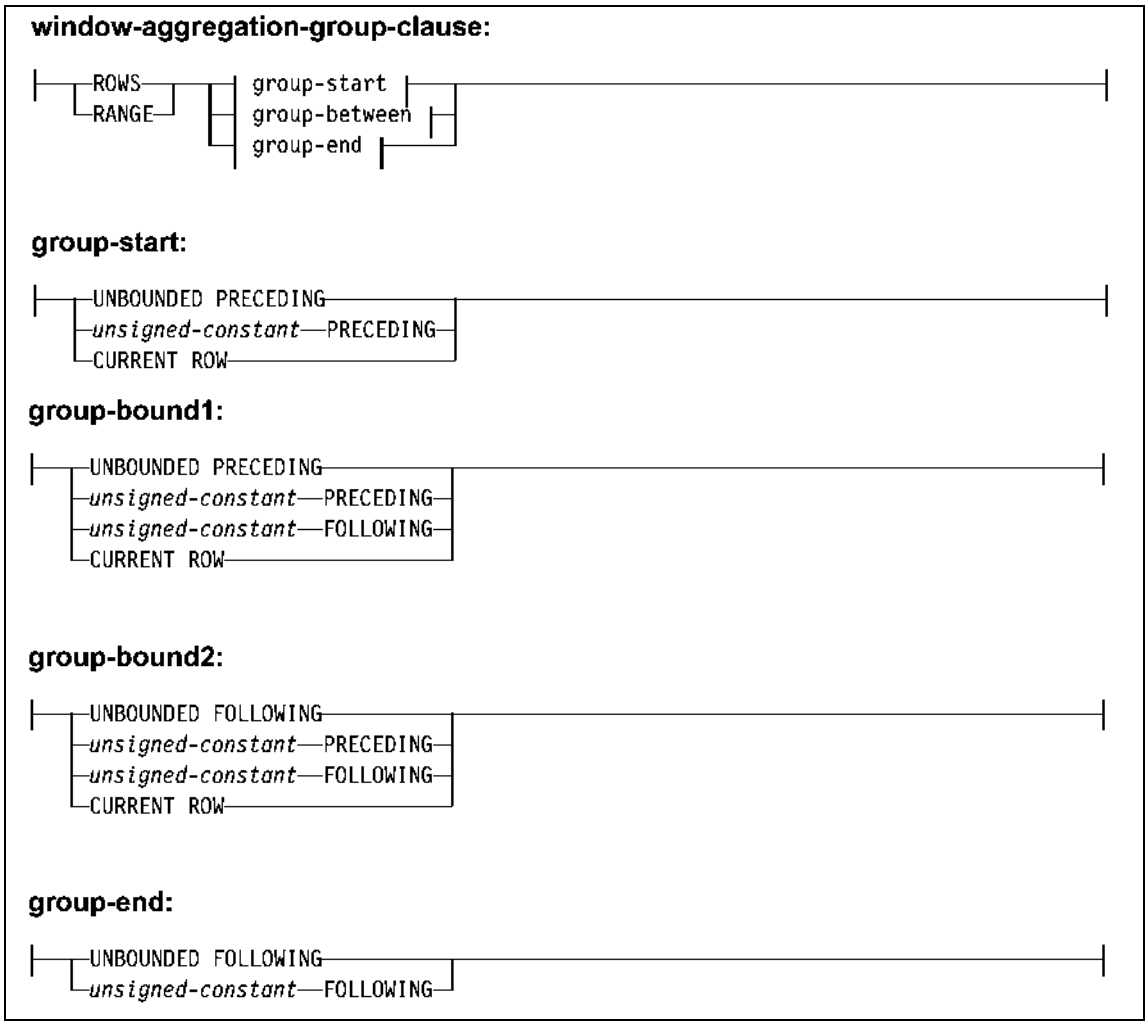


Figure 3-5 Window aggregation group clause

A brief explanation of some of the OLAP functions follows, with examples provided as appropriate.

RANK

The RANK function assigns a sequential rank of a row within a window.

The RANK of a row is defined as one plus the number of rows that strictly precede the row.

Rows that are not distinct within the ordering of the window are assigned equal ranks.

If two or more rows are not distinct with respect to the ordering, then there will be one or more gaps in the sequential rank numbering. That is, the results of RANK may have gaps in the numbers resulting from duplicate values.

DENSE_RANK

Like the RANK function, DENSE_RANK assigns a sequential rank to a row in a window. However, its DENSE_RANK is one plus the number of rows preceding it that are distinct with respect to the ordering. Therefore, there will be no gaps in the sequential rank numbering, with ties being assigned the same rank.

ROWNUMBER

ROWNUMBER computes the sequential row number of the row within the window defined by an ordering clause (if one is specified), starting with 1 for the first row and continuing sequentially to the last row in the window.

If an ordering clause, ORDER BY, is not specified in the window, the row numbers are assigned to the rows in arbitrary order as returned by the sub-select.

PARTITION BY

The PARTITION BY clause allows for subdividing the window into partitions. A *partitioning-expression* is used to define the partitioning of the result set.

ORDER BY

The ORDER BY clause defines the ordering of rows within a window that determines the value of the OLAP function or the meaning of the ROW values in the window-aggregation-group-clause (see the following section concerning the window-aggregation-group).

The ORDER BY clause does not define the ordering of the query result set.

A *sort-key-expression* is an expression used in defining the ordering of the rows within the window. This clause is required when using the RANK and DENSE_RANK functions.

There are two sorting sequences:

- ASC — Sorts the sort-key-expression in ascending order. Null values are considered last in the order by default since in DB2 nulls are considered high values.
- DESC — Sorts the sort-key-expression in descending order. Null values are considered first in the order unless NULLS LAST is specified.

Window aggregation group clause

The window-aggregation-group clause defines the window to a set of rows with a defined ordering relative to the rows in the window.

ROWS

ROWS indicates the window is defined by counting rows.

RANGE

RANGE indicates the window is defined by an offset from a sort key.

group-start, group-between and group-end

The group start, between and group-end functions define the ROWS or RANGE window to be some number of rows or range of rows around the current row in the window. These functions make it possible to compute moving average types of calculations.

group-start

Specifies the starting point for this aggregation group. The window ends at the current row when UNBOUNDED PRECEDING or PRECEDING is specified (more later). Specification of the group-start clause is the equivalent to a group-between clause of the form “BETWEEN group-start AND CURRENT ROW”.

group-between

Specifies the aggregation group start and end based on either ROWS or RANGE that fit within the specified group-bound1 (beginning) and group-bound2 (endpoint).

group-end

Specifies the ending point of the aggregation group. The aggregation group start is the current row. Specification of a group-end clause is the equivalent to a group-between clause of the form “BETWEEN CURRENT ROW AND group-end”.

Figure 3-6 graphically depicts the relationships among the various window bounds that follow.

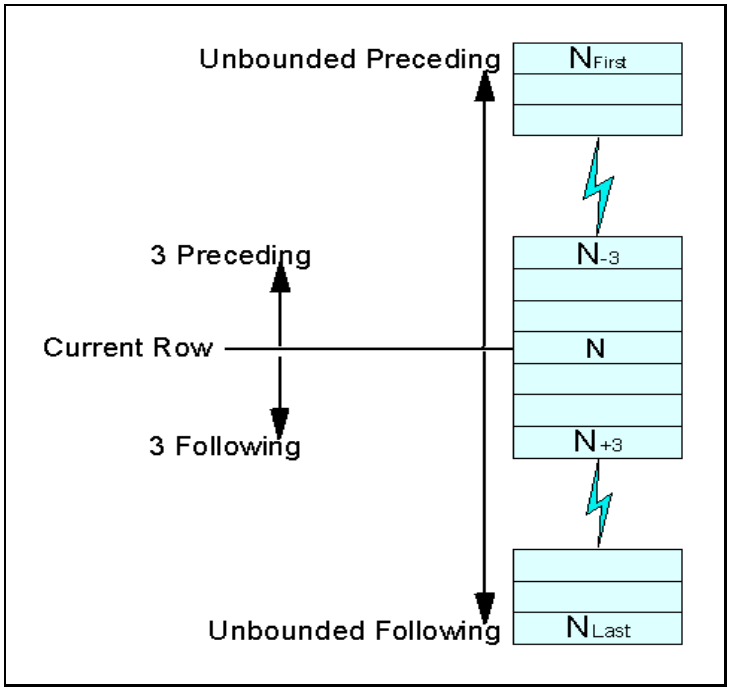


Figure 3-6 Windowing relationships

Note: N_{First} is always equal to 1.

Group-bounds one and two

A discussion of group-bounds follows.

- CURRENT ROW specifies the start or end of the window as the current row.
- UNBOUNDED PRECEDING includes the entire window preceding the current row. This can be specified with either ROWS or RANGE.
- UNBOUNDED FOLLOWING includes the entire window following the current row. This can be specified with either ROWS or RANGE.
- PRECEDING specifies either the range or number of rows preceding the current row as being in the window. If ROWS is specified, then *value* is a positive integer indicating a number of rows. If RANGE is specified, then the data type of *value* must be comparable to the type of the sort-key-expression of the window ORDER BY clause.

- FOLLOWING specifies either the range or number of rows following the current row as being in the window. If ROWS is specified, then *value* is a positive integer indicating a number of rows. If RANGE is specified, then the data type of *value* must be comparable to the type of the sort-key-expression of the window ORDER BY clause.

3.3.2 GROUPING capabilities ROLLUP & CUBE

The result of a GROUP BY operation is a set of groups of rows. Each row in this result represents the set of rows for which the grouping-expression is satisfied. Complex forms of the GROUP BY clause include grouping-sets and super-groups.

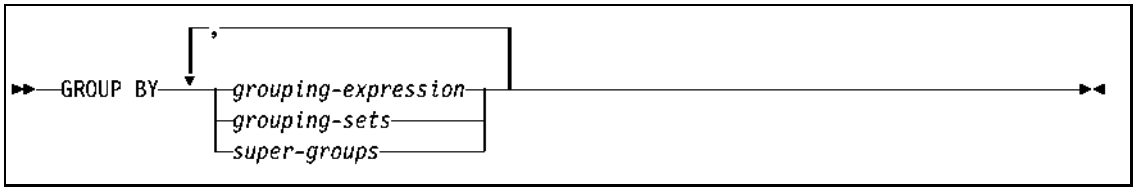


Figure 3-7 GROUP BY clause

Note: For grouping, all null values from a *grouping-expression* are considered equal.

A grouping sets specification allows multiple grouping clauses to be specified in a single statement. This can be thought of as a union of two or more groups of rows into a single result set. It is logically equivalent to the union of multiple subselects with the group by clause in each subselect corresponding to one grouping set. A grouping set can be a single element, or can be a list of elements delimited by parentheses, where an element is either a grouping expression, or a super-group.

```
GROUP BY a is equivalent to GROUP BY GROUPING SETS ((a))
GROUP BY a,b,c is equivalent to GROUP BY GROUPING SETS ((a,b,c))
```

In terms of OLAP functions we will confine our discussion to the two super-groups ROLLUP and CUBE, whose syntax is shown in Figure 3-8.

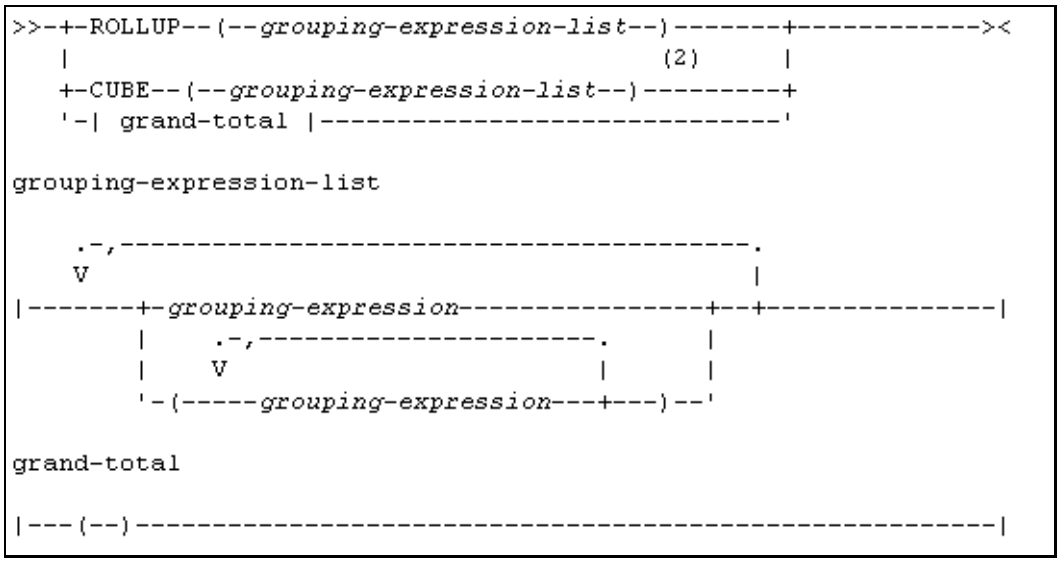


Figure 3-8 Super Groups ROLLUP & CUBE

ROLLUP

A ROLLUP group is an extension to the GROUP BY clause that produces a result set that contains *sub-total*[4] rows in addition to the 'regular' grouped rows. *Sub-total* rows are 'super-aggregate' rows that contain further aggregates whose values are derived by applying the same column functions that were used to obtain the grouped rows. A ROLLUP grouping is a series of *grouping-sets*.

For example:

```
GROUP BY ROLLUP (a,b,c)
is equivalent to
GROUP BY GROUPING SETS
(
(a,b,c)
(a,b)
(a)
()
)
```

Notice that the *n* elements of the ROLLUP translate to *n*+1 grouping sets.

[4] These are called sub-total rows, because that is their most common use. However, any column function can be used for the aggregation including MAX and AVG.

Note: The order in which the *grouping-expressions* is specified is significant for ROLLUP.

CUBE

The CUBE super-group is the other extension to the GROUP BY clause that produces a result set that contains all the sub-total rows of a ROLLUP aggregation and, in addition, contains 'cross-tabulation' rows. Cross-tabulation rows are additional 'super-aggregate' rows. They are, as the name implies, summaries across columns if the data were represented as a spreadsheet.

Like ROLLUP, a CUBE group can also be thought of as a series of grouping-sets. In the case of a CUBE, all permutations of the cubed grouping-expression are computed along with the grand total. Therefore, the "n" elements of a CUBE translate to 2^n (2 to the power n) grouping-sets. For instance, a specification of:

```
GROUP BY CUBE (a,b,c) is equivalent to
GROUP BY GROUPING SETS
(
(a,b,c)
(a,b)
(a,c)
(b,c)
(a)
(b)
(c)
()
)
```

Notice that the 3 elements of the CUBE translate to 8 grouping sets.

Unlike ROLLUP, the order of specification of elements does not matter for CUBE.

```
CUBE (DayOfYear, Sales_Person) is the same as CUBE (Sales_Person,DayOfYear)
```

CUBE is an extension of the Rollup clause. The CUBE clause not only provides the column summaries we saw in rollup but also calculates the row summaries and grand totals for the various dimensions.

3.3.3 Ranking, numbering, aggregation examples

These functions are useful in determining ranks, positioning, sequences and medians. They have been used by:

- Financial institutions to identify top profitable customers
- International Olympic Committee to rank contestants, assign medals and leading country medal rankings

Medians are useful in applications where the average is greatly influenced by a few extreme values, called outliers. Companies want to build sales campaigns that hit the largest segment of their target population and not the average.

Note: (The median is the midpoint of a set of data elements. Take following sequence of numbers: 3, 5, 7, 8, 37. The median is 7. The Average or mean is 12.

There is no DB2 function to compute the median of a set of data elements. However, the following SQL can be used to compute the median of a set of data elements.

Computing the median

The computation of the median value depends upon whether there are an odd number, or an even number of data points in the set.

- For an odd number, the median is the middle element of the sorted rows. In Example 3-12, the median value is 25.70.
- For an even number, the median is the average of the two middle elements of the sorted rows. In Example 3-13, the average of the sum of (25.7 + 32.0) = 28.85. There are other ways of dealing with an even number of data elements for example, return (n/2), or [(n+2) + 1] smallest value.

Example 3-12 Table containing an odd number of rows

```
1, 50.2
2, 25.7
3, 32.0
4, 17.2
5, 18.4
6, 19.6
7, 44.3
8, 22.5
9, 1000.7
```

Example 3-13 Table containing an even number of rows

```
1, 50.2
2, 25.7
3, 32.0
4, 17.2
5, 18.4
6, 19.6
7, 44.3
8, 1000.7
```

The SQL shown in Example 3-14 determines the median value for a table that may include either an odd number of rows, or an even number of rows. In the case of an odd number of rows, the virtual table ‘dt3’ has only a single row in it, while in the case of an even number of rows, ‘dt3’ has 2 rows in it. When the following query is issued against the table shown in Example 3-12, the median value is 25.70, and when executed against the table shown in Example 3-13, the median value is computed as 28.85.

Example 3-14 Compute median value with an even number of data points in the set

```
WITH
  dt1 AS (SELECT purchases, ROWNUMBER() OVER (ORDER BY purchases) AS num
          FROM cust_data),
  dt2 AS (SELECT COUNT(purchases) + 1 AS count FROM dt1),
  dt3 AS (SELECT purchases FROM dt1,dt2
          WHERE num = FLOOR(count/2e0) OR num = CEILING(count/2e0))
SELECT DECIMAL(AVG(purchases),10,2) AS median FROM dt3
```

Important: Note that there are no nulls in the set of data points. If nulls are present, then they should be *excluded* from the evaluation by adding to the above query a predicate such as ...WHERE purchases IS NOT NULL.

Note: It is worth observing that the value for average purchases is distorted by the customer making a purchase of 1000.70. This may or may not be a data entry error.

RANK example

Assume we would like to rank employees by total compensation and listed alphabetically.

The SQL in Example 3-15 shows that the ORDER BY clause in the RANK () OVER statement controls only the ranking sequence and not output sequence.

Note: When specifying an OLAP function like Rank, Dense_Rank, Row_Number, a window is specified that defines the rows over which the function is applied, and in what order. This window is specified via the OVER() clause.

Example 3-15 RANK() OVER example

```
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
FROM EMPLOYEE
WHERE SALARY+BONUS > 30000
ORDER BY LASTNAME
```

The result of this query is shown in Figure 3-9.

EMPNO	LASTNAME	FIRSTNME	TOTAL_SA...	RANK_SA...
000120	GEYER	JOHN	40775.00	6
000010	HAAS	CHRISTINE	53750.00	1
000120	HENDERS...	EILEEN	30350.00	11
000120	KWAN	SALLY	38850.00	7
000110	LUCCHESSI	VINCENZO	47400.00	3
000120	LUTZ	JENNIFER	30440.00	10
000120	PULASKI	EVA	36770.00	8
000120	STERN	IRVING	32850.00	9
999981	Symthe	Christine	53750.00	1
000120	THOMPSON	MICHAEL	41850.00	5
999982	Vino	Vincenzo	47400.00	3

Figure 3-9 Employee rank by total salary

One can see gaps and duplicates in the ranks in the above result. If DENSE_RANK is specified there will be no gaps in the sequential rank numbering.

DENSE_RANK example

Non-distinct rows get the same rank but the row following a tie is given the next sequential rank.

Rewriting the above SQL with DENSE_RANK would look as shown in Example 3-16.

Example 3-16 DENSE_RANK() OVER example

```
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
DENSE_RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
FROM EMPLOYEE
WHERE SALARY+BONUS > 30000
ORDER BY LASTNAME
```

The result of this query is shown in Figure 3-10.

EMPNO	LASTNAME	FIRSTNME	TOTAL_SA...	RANK_SA...
000120	GEYER	JOHN	40775.00	4
000010	HAAS	CHRISTINE	53750.00	1
000120	HENDERS...	EILEEN	30350.00	9
000120	KWAN	SALLY	38850.00	5
000110	LUCCHESSI	VINCENZO	47400.00	2
000120	LUTZ	JENNIFER	30440.00	8
000120	PULASKI	EVA	36770.00	6
000120	STERN	IRVING	32850.00	7
999981	Symthe	Christine	53750.00	1
000120	THOMPSON	MICHAEL	41850.00	3
999982	Vino	Vincenzo	47400.00	2

Figure 3-10 Employee DENSE_RANK by total salary

[2]

Note: Since Nulls collate high, nulls in RANK and DENSERANK functions are ranked first for descending rankings. This can be overridden with the “nulls last” parameter, RANK () OVER (ORDER BY salary desc nulls last) as ranking.

ROW_NUMBER, RANK and DENSE_RANK example

Since ROW_NUMBER creates a sequential numbering to the rows in the window, it can be used with an ORDER BY clause in the window to eliminate gaps or duplicates.

Without the ORDER BY clause, ROW_NUMBER assigns sequential numbers to rows arbitrarily as retrieved by the subselect. Such a result is not related to ranking but merely assigning arbitrary numbers to rows.

Example 3-17 shows the differences between RANK, DENSE_RANK and ROW_NUMBER.

Example 3-17 ROW_NUMBER, RANK, DENSE_RANK example

```
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY,
DENSE_RANK() OVER (ORDER BY SALARY+BONUS DESC) AS DENSERANK,
ROW_NUMBER() OVER (ORDER BY SALARY+BONUS DESC) AS ROW_NUMBER
FROM EMPLOYEE
WHERE SALARY+BONUS > 30000
```

The result of this query is shown in Figure 3-11.

EMPNO	LASTNAME	FIRSTNME	TOTAL_SALARY	RANK_SALARY	DENSERANK	ROW_NUMBER
000010	HAAS	CHRISTINE	53750.00	1	1	1
999981	Symthe	Christine	53750.00	1	1	2
000110	LUCCHESSI	VINCENZO	47400.00	3	2	3
999982	Vino	Vincenzo	47400.00	3	2	4
000120	THOMPSON	MICHAEL	41850.00	5	3	5
000120	GEYER	JOHN	40775.00	6	4	6
000120	KWAN	SALLY	38850.00	7	5	7
000120	PULASKI	EVA	36770.00	8	6	8
000120	STERN	IRVING	32850.00	9	7	9
000120	LUTZ	JENNIFER	30440.00	10	8	10
000120	HENDERS...	EILEEN	30350.00	11	9	11

Figure 3-11 RANK, DENSE_RANK and ROW_NUMBER comparison

RANK and PARTITION BY example

The following is an example of using the PARTITION BY clause which allows for subdividing the rows into partitions. It functions similar to the GROUP BY function, but is local to the window set whereas GROUP BY is a global function.

Assume we want to find the top 4 ranking of employee salary within each department.

We will need to use the RANK function with partition (ranking window) by department. We will need to use a common table expression otherwise the reference to RANK_IN_DEPT in our subselect is ambiguous.

The SQL is shown in Example 3-18.

Example 3-18 RANK & PARTITION example

```
WITH SALARYBYDEPT AS(
   SELECT WORKDEPT, LASTNAME, FIRSTNME,SALARY,
      RANK() OVER (PARTITION BY WORKDEPT ORDER BY SALARY DESC NULLS LAST)
      AS RANK_IN_DEPT
   FROM EMPLOYEE )
SELECT WORKDEPT, LASTNAME,FIRSTNME,SALARY, RANK_IN_DEPT
FROM SALARYBYDEPT
WHERE RANK_IN_DEPT <= 4 AND WORKDEPT IN
('A00','A11','B01','C01','D1','D11')
ORDER BY WORKDEPT, RANK_IN_DEPT, LASTNAME
```

The result of this query is shown in Figure 3-12.

WORKDEPT	LASTNAME	FIRSTNME	SALARY	RANK_IN_DEPT
A00	HAAS	CHRISTINE	52750.00	1
A00	LUCCHESSI	VINCENZO	46500.00	2
A00	O'CONNELL	SEAN	29250.00	3
A11	Symthe	Christine	52750.00	1
A11	Vino	Vincenzo	46500.00	2
A11	Walker	Sean	29250.00	3
B01	THOMPSON	MICHAEL	41250.00	1
C01	KWAN	SALLY	38250.00	1
C01	NICHOLLS	HEATHER	28420.00	2
C01	QUINTANA	DOLORES	23800.00	3
D1	Homemaker	Sally	60000.00	1
D1	Bookem	Dan	50000.00	2
D1	Willie	Samantha	48000.00	3
D1	Begood	John	45000.00	4
D1	Dancing	Jean	45000.00	4
D11	STERN	IRVING	32250.00	1
D11	LUTZ	JENNIFER	29840.00	2
D11	BROWN	DAVID	27740.00	3
D11	ADAMSON	BRUCE	25280.00	4

Figure 3-12 PARTITION BY window results

The employee table is first partitioned by department. Then the ranking function is applied based on highest to lowest salary within the common table expression. Then the outer select chooses only the top 4 employees in the departments requested and orders them by department and rank in the department. Ties in rank are listed alphabetically.

OVER clause example

With the OVER clause it is possible to turn aggregate function like SUM, AVG, COUNT, COUNT_BIG, CORRELATION, VARIANCE, COVARIANCE, MIN, MAX, and STDDEV into OLAP functions. Rather than returning the aggregate of the rows as a single value the OVER function operates on the range of rows specified in the window and returns a single aggregate value for the range. The following example illustrates this function.

Assume we would like to determine for each employee within a department the percentage of that employee's salary to the total department salary. That is if an employee's salary is $20,000 and the department total is $100,000 then the employee's percentage of the department's salary is 20%.

Our SQL would look as shown in Example 3-19:

Example 3-19 OVER clause example

```
SELECT WORKDEPT,LASTNAME,SALARY, DECIMAL(SALARY,15,0)*100/SUM(SALARY)
OVER (PARTITION BY WORKDEPT) AS DEPT_SALARY_PERCENT
FROM EMPLOYEE
WHERE WORKDEPT IN ('A00','A11','B01','C01','D1','D11')
ORDER BY WORKDEPT, DEPT_SALARY_PERCENT DESC
```

The result of this query is shown in Figure 3-13. The SUM(SALARY) (sum of salary) is ranged by the OVER (PARTITION BY... clause to only those values in each department.

WORKDEPT	LASTNAME	SALARY	DEPT_SALARY_PERCENT
A00	HAAS	52750.00	41.050
A00	LUCCHESSI	46500.00	36.186
A00	O'CONNELL	29250.00	22.762
A11	Symthe	52750.00	41.050
A11	Vino	46500.00	36.186
A11	Walker	29250.00	22.762
B01	THOMPSON	41250.00	100.000
C01	KWAN	38250.00	42.279
C01	NICHOLLS	28420.00	31.413
C01	QUINTANA	23800.00	26.307
D1	Homemaker	60000.00	13.636
D1	Bookem	50000.00	11.363
D1	Willie	48000.00	10.909
D1	Dancing	45000.00	10.227
D1	Begood	45000.00	10.227
D1	Knowsit	44000.00	10.000
D1	Notdancing	43000.00	9.772
D1	Maker	40000.00	9.090
D1	Baker	35000.00	7.954
D1	James	30000.00	6.818
D11	STERN	32250.00	14.520
D11	LUTZ	29840.00	13.435
D11	BROWN	27740.00	12.489
D11	ADAMSON	25280.00	11.382
D11	YOSHIMURA	24680.00	11.112
D11	PIANKA	22250.00	10.018
D11	SCOUTTEN	21340.00	9.608
D11	WALKER	20450.00	9.207
D11	JONES	18270.00	8.226

Figure 3-13 Salary as a percentage of department total salary

This same concept can be applied to determine product percentage of sales for various product groups within a retail store, bank or distribution center.

ROWS and ORDER BY example

It is possible to define the rows in the window function using a window aggregate clause when an ORDER BY clause is included in the definition. This allows the inclusion or exclusion of ranges of values or rows within the ordering clause. Assume we want to smooth the curve of random data similar to the 50 and 200 day moving average of stock price found on numerous stock Web sites.

The SQL and the result of the query are shown in Example 3-20 and Figure 3-14.

Example 3-20 ROWS & ORDER BY example

```
SELECT DATE,SYMBOL,CLOSE_PRICE,AVG(CLOSE_PRICE) OVER
(ORDER BY DATE ROWS 5 PRECEDING) AS SMOOTH
FROM STOCKTAB
WHERE SYMBOL = 'IBM'
```

DATE	SYMBOL	CLOSE_P...	SMOOTH
1999-08-02	IBM	110.125	110.12500...
1999-08-03	IBM	109.500	109.81250...
1999-08-04	IBM	112.000	110.54166...
1999-08-05	IBM	110.625	110.56250...
1999-08-06	IBM	112.750	111.00000...
1999-08-09	IBM	110.625	111.10000...
1999-08-10	IBM	108.375	110.87500...
1999-08-11	IBM	109.250	110.32500...
1999-08-12	IBM	109.375	110.07500...
1999-08-13	IBM	108.500	109.22500...
1999-08-16	IBM	110.250	109.15000...
1999-08-17	IBM	108.375	109.15000...
1999-08-18	IBM	108.375	108.97500...
1999-08-19	IBM	109.375	108.97500...
1999-08-20	IBM	112.000	109.67500...
1999-08-23	IBM	113.125	110.25000...
1999-08-24	IBM	114.875	111.55000...
1999-08-25	IBM	115.500	112.97500...
1999-08-26	IBM	113.375	113.77500...
1999-08-27	IBM	115.625	114.50000...
1999-08-30	IBM	113.625	114.60000...
1999-08-31	IBM	112.875	114.20000...
1999-09-01	IBM	115.625	114.22500...

Figure 3-14 Five day smoothing of IBM

Note: IBM stock price data is not historically accurate.

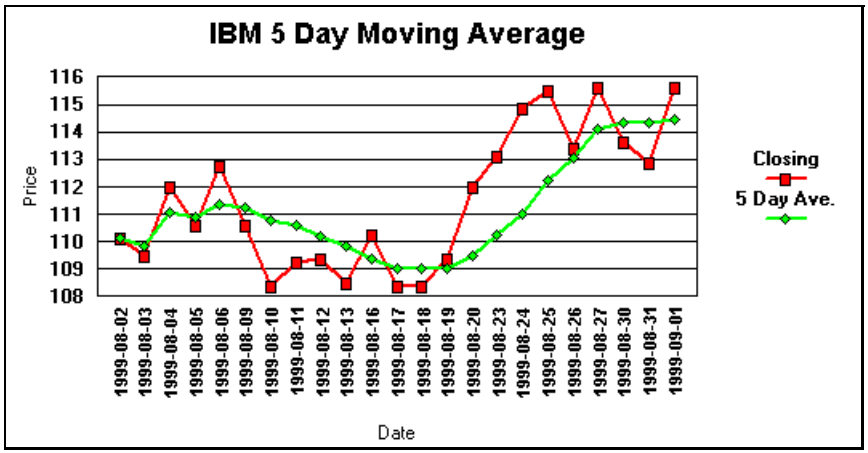


Figure 3-15 IBM five day moving average

The equivalent result can be calculated using the RANGE instead of ROWS. ROWS works well in situations when the data is dense, that is, there are no values duplicated or missing.

ROWS, RANGE, and ORDER BY example

Stock tables have the weekends missing. RANGE can be used to overcome gaps as illustrated in the following example.

Assume we want to calculate the seven day calendar average with the intent of taking into account the weekends. We will compare the results of ROWS versus RANGE. The SQL is shown in Example 3-21:

Example 3-21 ROWS, RANGE & ORDER BY example

```
SELECT DATE, SUBSTR(DAYNAME(DATE),1,9) AS DAY_WEEK, CLOSE_PRICE,
DEC(AVG(CLOSE_PRICE) OVER (ORDER BY DATE ROWS 6 PRECEDING),7,2) AS
AVG_7_ROWS,
COUNT(CLOSE_PRICE) OVER (ORDER BY DATE ROWS 6 PRECEDING) AS COUNT_7_ROWS,
DEC(AVG(CLOSE_PRICE) OVER (ORDER BY DATE RANGE 00000006. PRECEDING),7,2) AS
AVG_7_RANGE,
COUNT(CLOSE(CLOSE_PRICE) OVER (ORDER BY DATE RANGE 00000006. PRECEDING) AS
COUNT_7_RANGE
FROM STOCKTAB
WHERE SYMBOL='IBM'
```

Note: The result of a DATE arithmetic operation is a DEC(8,0) value. We therefore need to specify the comparison value in the RANGE operator with a precision of DEC(8,0), in order to obtain the correct result.

The result of this query is shown in Figure 3-16 , and it illustrates the difference in ROWS versus RANGE.

Attempting to use ROWS in setting the window for seven calendar days actually returns 7 preceding rows. These seven rows span more than one calendar week.

RANGE fixes this problem by recognizing the weekend gap. Therefore RANGE is appropriate when there are gaps in the input data.

DATE	DAY_WEEK	CLOSE_P...	AVG_7_R...	COUNT_7...	AVG_7_RA...	COUNT_7...
1999-08-02	Monday	110.125	110.12	1	110.12	1
1999-08-03	Tuesday	109.500	109.81	2	109.81	2
1999-08-04	Wednesday	112.000	110.54	3	110.54	3
1999-08-05	Thursday	110.625	110.56	4	110.56	4
1999-08-06	Friday	112.750	111.00	5	111.00	5
1999-08-09	Monday	110.625	110.93	6	111.10	5
1999-08-10	Tuesday	108.375	110.57	7	110.87	5
1999-08-11	Wednesday	109.250	110.44	7	110.32	5
1999-08-12	Thursday	109.375	110.42	7	110.07	5
1999-08-13	Friday	108.500	109.92	7	109.22	5
1999-08-16	Monday	110.250	109.87	7	109.15	5
1999-08-17	Tuesday	108.375	109.25	7	109.15	5
1999-08-18	Wednesday	108.375	108.92	7	108.97	5
1999-08-19	Thursday	109.375	109.07	7	108.97	5
1999-08-20	Friday	112.000	109.46	7	109.67	5
1999-08-23	Monday	113.125	110.00	7	110.25	5
1999-08-24	Tuesday	114.875	110.91	7	111.55	5
1999-08-25	Wednesday	115.500	111.66	7	112.97	5
1999-08-26	Thursday	113.375	112.37	7	113.77	5
1999-08-27	Friday	115.625	113.41	7	114.50	5
1999-08-30	Monday	113.625	114.01	7	114.60	5
1999-08-31	Tuesday	112.875	114.14	7	114.20	5
1999-09-01	Wednesday	115.625	114.50	7	114.22	5

Figure 3-16 Seven calendar day moving average

3.3.4 GROUPING, GROUP BY, ROLLUP and CUBE examples

We provide examples using GROUPING, GROUP BY, ROLLUP and CUBE.

GROUPING, GROUP BY and CUBE example

Grouping is used in conjunction with the super-group functions, GROUP BY CUBE or ROLLUP. The purpose of the GROUPING function is to identify summary rows in the CUBE and ROLLUP query results. The GROUPING function returns a one or a zero to indicate whether or not a row returned by the GROUP BY function is a sub-total row generated by the GROUP BY function.

A one means the row was the result of a sub-total, and a zero means the row was not the result of a sub-total.

The input to the GROUPING function can be any type, but must be an item of the associated GROUP BY clause. Consider Example 3-22.

Example 3-22 GROUPING, GROUP BY & CUBE example

```
SELECT SALES_DATE,
         SALES_PERSON,
         SUM(SALES) AS UNITS_SOLD,
         GROUPING(SALES_DATE) AS DATE_GROUP,
         GROUPING(SALES_PERSON) AS  SALES_GROUP
FROM SALES
GROUP BY CUBE (SALES_DATE, SALES_PERSON)
ORDER BY SALES_DATE, SALES_PERSON
```

The result of this query is shown in Figure 3-17.

SALES_DATE	SALES_PERSON	UNITS_SOLD	DATE_GROUP	SALES_GROUP
12/31/1995	GOUNOT	1	0	0
12/31/1995	LEE	6	0	0
12/31/1995	LUCCHESSI	1	0	0
12/31/1995	-	8	0	1
03/29/1996	GOUNOT	11	0	0
03/29/1996	LEE	12	0	0
03/29/1996	LUCCHESSI	4	0	0
03/29/1996	-	27	0	1
03/30/1996	GOUNOT	21	0	0
03/30/1996	LEE	21	0	0
03/30/1996	LUCCHESSI	4	0	0
03/30/1996	-	46	0	1
03/31/1996	GOUNOT	3	0	0
03/31/1996	LEE	27	0	0
03/31/1996	LUCCHESSI	1	0	0
03/31/1996	-	31	0	1
04/01/1996	GOUNOT	14	0	0
04/01/1996	LEE	25	0	0
04/01/1996	LUCCHESSI	4	0	0
04/01/1996	-	43	0	1
-	GOUNOT	50	1	0
-	LEE	91	1	0
-	LUCCHESSI	14	1	0
-	-	155	1	1

Figure 3-17 Grouping result

Note: Figure 3-17 is output from the DB2 Command Line Processor. Here nulls are represented as “-”.

The ‘1’s in the DATE_GROUP column indicate the value in the UNIT_SOLD column are sub-total rows generated by the GROUP BY CUBE clause. Likewise the ones in SALES_GROUP column indicate these rows are also sub-total rows. The last row were DATE_GROUP and SALES_GROUP are both one indicates this row is a grand total row.

This function is used for end user applications built to recognize SALES_DATE sub-total row by the fact that the value of DATE_GROUP is 0, and the value of SALES_GROUP is 1.

A SALES_PERSON sub-total row can be recognized by the fact that the value of DATE_GROUP is 1 and the value of SALES_GROUP is 0. A grand total row can be recognized by the value 1 for both DATE_GROUP and SALES_GROUP.

ROLLUP example

In our sales data example in Figure 3-18 and Figure 3-19, we want to summarize the sales data by sales person and date with a rollup of sales to a day and week level for weeks 13 and 14 in 1996.

SALES_DATE	SALES_PERSON	REGION	SALES
1996-03-29	LUCCHESSI	Ontario-So...	3
1996-03-29	LUCCHESSI	Quebec	1
1996-03-29	LEE	Ontario-So...	2
1996-03-29	LEE	Ontario-No...	2
1996-03-29	LEE	Quebec	3
1996-03-29	LEE	Manitoba	5
1996-03-29	GOUNOT	Ontario-So...	3
1996-03-29	GOUNOT	Quebec	1
1996-03-29	GOUNOT	Manitoba	7
1996-03-30	LUCCHESSI	Ontario-So...	1
1996-03-30	LUCCHESSI	Quebec	2
1996-03-30	LUCCHESSI	Manitoba	1
1996-03-30	LEE	Ontario-So...	7
1996-03-30	LEE	Ontario-No...	3
1996-03-30	LEE	Quebec	7
1996-03-30	LEE	Manitoba	4
1996-03-30	GOUNOT	Ontario-So...	2
1996-03-30	GOUNOT	Quebec	18
1996-03-30	GOUNOT	Manitoba	1
1996-03-31	LUCCHESSI	Manitoba	1
1996-03-31	LEE	Ontario-So...	14
1996-03-31	LEE	Ontario-No...	3
1996-03-31	LEE	Quebec	7
1996-03-31	LEE	Manitoba	3
1996-03-31	GOUNOT	Ontario-So...	2
1996-03-31	GOUNOT	Quebec	1

Figure 3-18 Sales item detail for March

SALES_DA...	SALES_PE...	REGION	SALES
1996-04-01	LUCCHESSI	Ontario-So...	3
1996-04-01	LUCCHESSI	Manitoba	1
1996-04-01	LEE	Ontario-So...	8
1996-04-01	LEE	Ontario-No...	
1996-04-01	LEE	Quebec	8
1996-04-01	LEE	Manitoba	9
1996-04-01	GOUNOT	Ontario-So...	3
1996-04-01	GOUNOT	Ontario-No...	1
1996-04-01	GOUNOT	Quebec	3
1996-04-01	GOUNOT	Manitoba	7
1996-04-05	LEE	Manitoba	5
1996-04-05	LEE	Quebec	3
1996-04-05	LEE	Ontario-So...	3
1996-04-05	LEE	Ontario-No...	1
1996-04-06	LEE	Manitoba	3
1996-04-06	LEE	Quebec	2
1996-04-06	LEE	Ontario-So...	1
1996-04-06	LEE	Ontario-No...	1
1996-04-06	LUCCHESSI	Manitoba	2
1996-04-06	LUCCHESSI	Quebec	2
1996-04-06	LUCCHESSI	Ontario-So...	2
1996-04-06	LUCCHESSI	Ontario-No...	2
1996-04-05	LUCCHESSI	Manitoba	1
1996-04-05	LUCCHESSI	Quebec	1
1996-04-05	LUCCHESSI	Ontario-So...	2
1996-04-05	LUCCHESSI	Ontario-No...	1
1996-04-05	GOUNOT	Manitoba	5

Figure 3-19 Sales item detail for April

Our SQL looks as shown in Example 3-23.

Example 3-23 ROLLUP example

```
SELECT  WEEK(SALES_DATE) AS WEEK,
        DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
        SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) IN (13,14) AND
YEAR(SALES_DATE) = 1996
GROUP BY ROLLUP ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

The results are presented in Figure 3-20.

.

WEEK	DAY_WEEK	SALES_PERSON	UNITS_SOLD
13	6	GOUNOT	11
13	6	LEE	12
13	6	LUCCHESSI	4
13	6		27
13	7	GOUNOT	21
13	7	LEE	21
13	7	LUCCHESSI	4
13	7		46
13			73
14	1	GOUNOT	3
14	1	LEE	27
14	1	LUCCHESSI	1
14	1		31
14	2	GOUNOT	14
14	2	LEE	25
14	2	LUCCHESSI	4
14	2		43
14	6	GOUNOT	5
14	6	LEE	12
14	6	LUCCHESSI	5
14	6		22
14	7	LEE	7
14	7	LUCCHESSI	8
14	7		15
14			111
			184

Figure 3-20 Results of the ROLLUP query

Note: The last row in Figure 3-20 has no entry in the first two columns. These blanks are technically speaking nulls. The DB2 Command Center translates nulls to blanks in this case. Other tools may display nulls differently. This same behavior is seen in the DB2 Command Center output for CUBE.

The key to translating the format of the query results is to recognize the output format is controlled by the ORDER BY statement. In the preceding example, the output is sequenced first on week, then days within that week and finally by sales person for that day. Secondly, a summary or rollup row is inserted based on the order of the rollup statement. It is processed in reverse order. First the rollup for each sales person is given for the first day. Then, for that day, a rollup is given.

After all sales person summaries for that day are presented. After all days in a week are processed in this manner then a rollup row for each week is given. This process continues until all weeks are processed. Finally, a rollup grand total is given.

To put in succinctly, rollup processing provides column summaries. This is demonstrated in Figure 3-21.

The results are best viewed as tables, spreadsheets or bar charts. These tables and charts can be made by translating the preceding results into and commercially available spreadsheet and charting tool. These are shown in Figure 3-22 and Figure 3-23.

Week 14					
	Day 1	Day 2	Day 6	Day 7	
Gounot	3	14	5	0	
Lee	27	25	12	7	
Lucch				8	
Subto				15	
Total					74

Week 13			
	Day 6	Day 7	
Gounot	11	21	
Lee	12	21	
Lucchessi	4	4	
Subtotal	27	46	
Total			73

Figure 3-21 ROLLUP visualization as tables

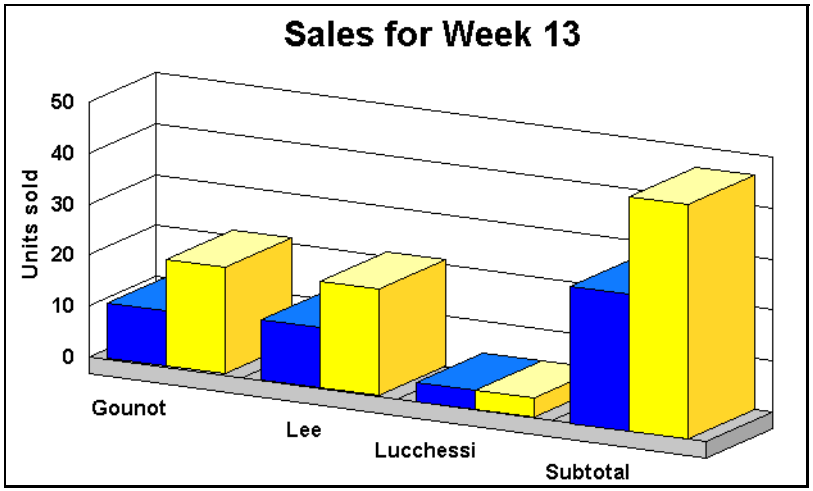


Figure 3-22 ROLLUP visualization as bar chart - week 13

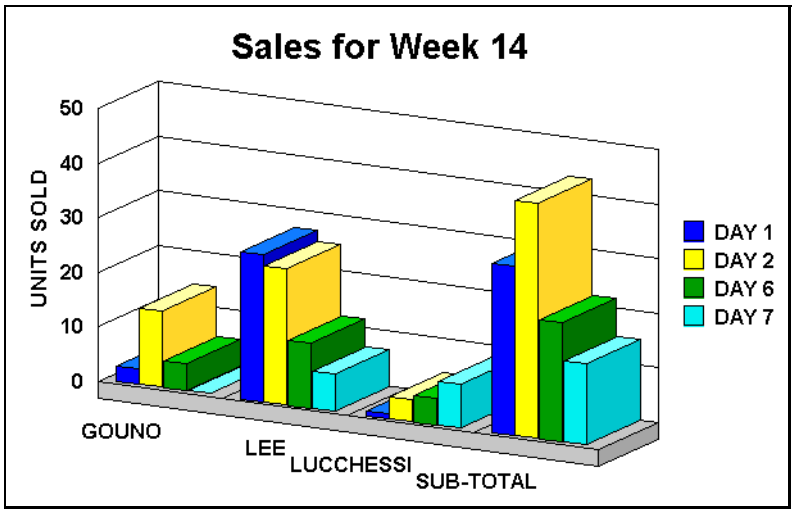


Figure 3-23 ROLLUP visualization as bar chart - week 14

CUBE example

To calculate a CUBE in our previous example we merely replace ROLLUP with CUBE in the GROUP BY clause as shown in Example 3-24.

Example 3-24 CUBE example

```
SELECT  WEEK(SALES_DATE) AS WEEK,
        DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
        SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) in (13,14) AND
YEAR(SALES_DATE) = 1996
GROUP BY CUBE ( WEEK(SALES_DATE),
DAYOFWEEK(SALES_DATE), SALES_PERSON )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

The result of this query is shown in Figure 3-24.

WEEK	DAY_WEEK	SALES_PERSON	UNITS_SOLD
13	6	GOUNOT	11
13	6	LEE	12
13	6	LUCCHESSI	4
13	6		27
13	7	GOUNOT	21
13	7	LEE	21
13	7	LUCCHESSI	4
13	7		46
13		GOUNOT	32
13		LEE	33
13		LUCCHESSI	8
13			73
14	1	GOUNOT	3
14	1	LEE	27
14	1		31
14	2	GOUNOT	14
14	2	LEE	25
14	2	LUCCHESSI	4
14	2		43
14	6	GOUNOT	5
14	6	LEE	12
14	6	LUCCHESSI	5
14	6		22
14	7	LEE	7
14	7	LUCCHESSI	8
14	7		15
14		GOUNOT	22
14		LEE	71
14		LUCCHESSI	18
14			111
	1	GOUNOT	3
	1	LEE	27
	1	LUCCHESSI	1
	1		31
	2	GOUNOT	14
	2	LEE	25
	2	LUCCHESSI	4
	2		43
	6	GOUNOT	16
	6	LEE	24
	6	LUCCHESSI	9
	6		49
	7	GOUNOT	21
	7	LEE	28
	7	LUCCHESSI	12
	7		61
		GOUNOT	54
		LEE	104
		LUCCHESSI	26
			184

Figure 3-24 CUBE query result

These results are more readily understood when translated into a three dimensional cube (our three dimensions are Weeks, Days and Sales Person) or tables laid out one on top of another.

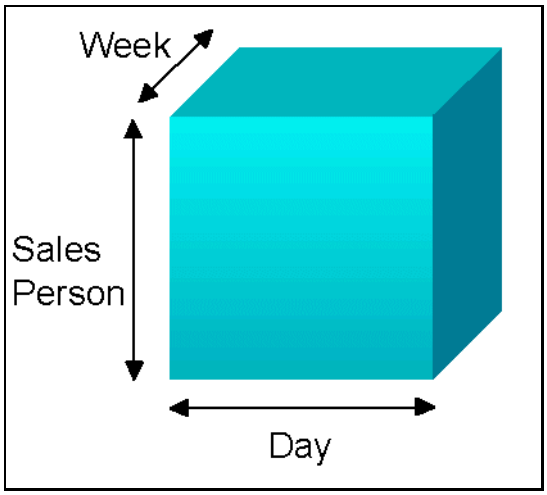


Figure 3-25 Three dimensional cube - sales by sales person, day, week

In Figure 3-26 we have added labels to the sections of the query result to aid in the creation of the tables. Unlike the results in Rollup, which are column summary tables, the results of a Cube are cross tabulation tables. The three tables based upon our example are:

- Units Sold for Sales Person by Day for Week 13
- Units Sold for Sales Person by Day for Week 14
- Units Sold - Sales Person by Day for Weeks 13 and 14

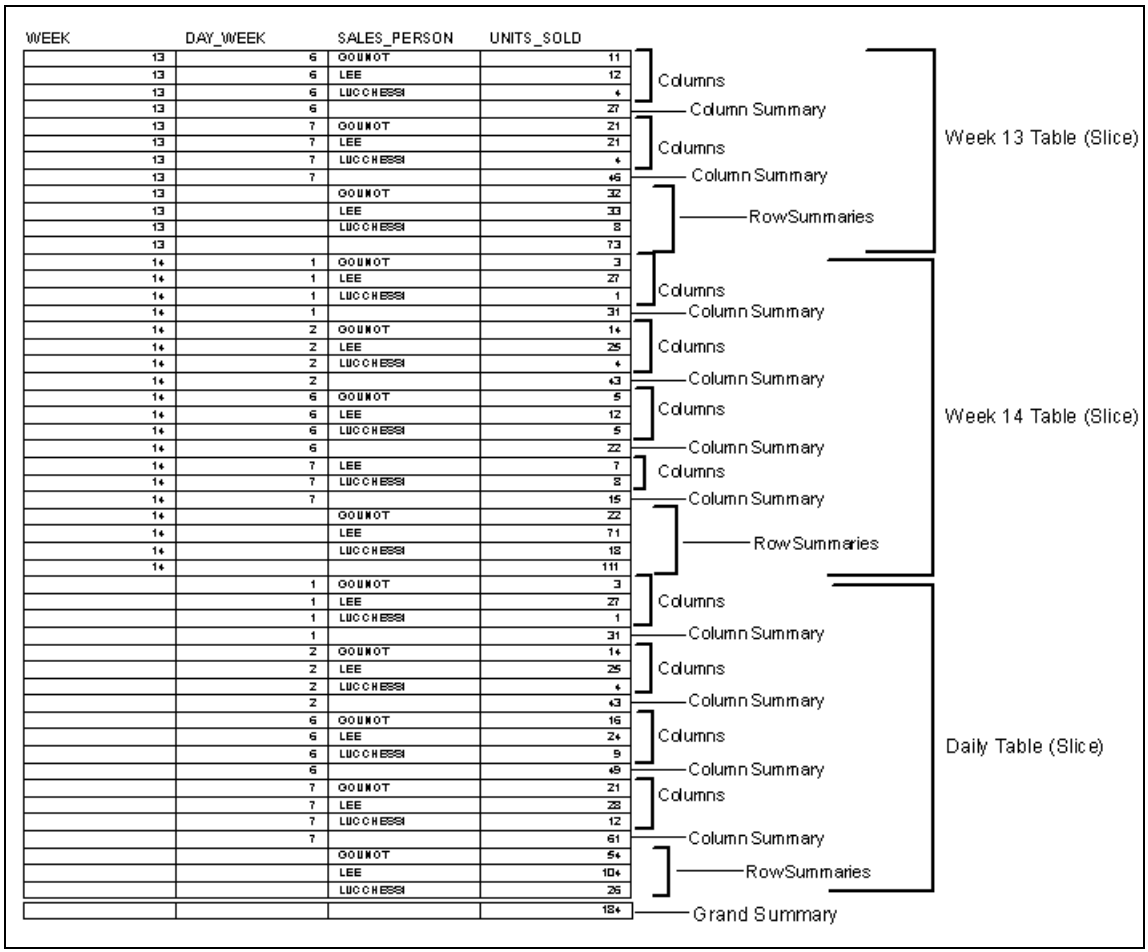


Figure 3-26 CUBE query result explanation

The tables are built using any commercially available spreadsheet tool. We simply follow the template provided in Figure 3-26 to create the three tables in Figure 3-27. In Figure 3-27 the super-aggregate rows are represented as the *column* labeled “Total” in those tables. The *rows* labeled “Total” in those tables are the row summaries referred to in the previous discussion of sub-total rows for the ROLLUP function.

Week 13/14	Day 1	Day 2	Day 6	Day 7	Total
Gounot	3	14	16	21	54
Lee	27	25	24	28	104
Lucchessi	1	4	9	12	26
					184

Week 14	Day 1	Day 2	Day 6	Day 7	Total
Gounot	3	14	5	-	22
Lee	27	25	12	7	71
Lucchessi	1	4	5	8	18
Total	31	43	22	15	111

Week 13	Day 1	Day 2	Day 6	Day 7	Total
Gounot	-	-	11	21	32
Lee	-	-	12	21	33
Lucchessi	-	-	4	4	8
Total	-	-	27	46	73

Figure 3-27 CUBE query tables

4

Statistics, analytic, OLAP functions in business scenarios

In this chapter we discuss typical business level queries that can be answered using DB2 UDB's statistics, analytic and OLAP functions. These business queries are categorized by industry, and describe the steps involved in resolving the query, with sample SQL and visualization of the results.

The target audience is application developers, and DBAs who are unfamiliar with statistics, analytic and OLAP functions, but quite knowledgeable about SQL.

4.1 Introduction

We describe typical business queries in the following industries:

- Retail
- Finance
- Sports

Attention: Many of these business queries problems apply to other industries as well

This is the format we followed for each industry sector:

1. A statement of the business requirement
2. Data fields/attributes required to address the requirement
3. DB2 UDB functions used in the queries
4. The steps involved in addressing the requirement, and for each step:
   - The SQL required
   - The result set
   - How the result set is converted for visualization
   - Visualization of the result

Important: Source data to answer the business query varies from organization to organization. We therefore assume that appropriate extraction, cleansing, and transformation of the data has been done to present it in one or more normalized tables for our queries. The latency of the data and the end-to-end performance perception of the business query is impacted by the efficiency of the extraction and transformation process.

The data for our queries come from different sources, and some of these are documented in Appendix B, “Tables used in the examples” on page 235, while the content of others are listed in the business requirement solution.

4.1.1 Using sample data

In many cases, the volume of data available may be very large, and it may not be cost-effective or timely enough to analyze the entire data. In such cases, it would be appropriate to take a representative sample of the data and perform analyses on it instead. An efficient and cost-effective sampling function can have a significant impact on the scalability of a system involving large volumes of data that typify the e-business environment.

DB2 provides support for a RAND function which uses a “Bernoulli Sampling” technique. See Chapter 3.2.8, “RAND” on page 104 for an explanation of the RAND function.

The quality of the sample and the size of the sample will play a significant role in the accuracy of the result obtained. A discussion of these considerations is beyond the scope of this document, suffice to say that these factors are unique to each domain and possibly to each organization.

Attention: The reader is strongly urged to get familiar with sampling theory and its applicability to their business environment prior to using DB2 UDB’s RAND function.

Important: When sampling from a single table, care should be taken to ensure that the extracted sample is representative, and large enough to provide an acceptable degree of accuracy. Trial and error is probably the best practical approach to hone in on an acceptable sample size.

Another factor to be considered is that in general, a join of sampled tables is not statistically equivalent to a sample from the join of the original tables. An acceptable approach for a join involving referentially constrained tables, may be to sample the foreign key table, and then extract the rows in the referenced table using the foreign key values in the sample. You should evaluate the efficacy of this approach in your particular environment.

Sampling may be used for auditing, data mining as well as getting approximate answers to aggregation type questions.

4.1.2 Sampling and aggregation example

Assume we have a very large table containing sales data by country and we would like to obtain the sales summary by year and country using sampling, and assess the “standard error” of the estimate. We use the familiar trans, transitem, and loc tables for this query. Consider Example 4-2.

Example 4-1 Sample & aggregation example

```
SELECT loc.country AS country, YEAR(t.pdate) AS year,
  SUM(ti.amount) / :samp_rate AS est_sales,
  SQRT((1e0/:samp_rate)*(1e0-(1e0/:samp_rate))*SUM(amount*amount)) AS
std_err
FROM trans t, transitem ti, loc loc
WHERE t.transid = ti.transid AND loc.locid = t.locid
      AND RAND(1) < :samp_rate
GROUP BY loc.country, YEAR(t.pdate)
```

Notes: The samp_rate is a parameter that specifies the sampling rate. In the foregoing example, our sampling rate is 0.01 or 1%.

Also, the formula used in the query for std_err computation is offered without explanation.

The foregoing query takes a Bernoulli sample from a join of the 3 tables, using a sampling rate of 0.01 resulting in approximately 1% of the rows being selected. In order to estimate the sum for the entire table, we need to **scale the answer up** by a factor of (1 / :samp_rate) which is 100. The “standard error” is computed as shown.

The result of the foregoing query is as follows:

```
country    year  est_sales  std_err
-------  ------  ---------  -------
    USA    1998     127505  1326.09
    USA    1999     236744  2133.17
GERMANY    1998      86278   961.45
GERMANY    1999     126737  1488.66
...
```

Typically, there is a high probability that the true sum will lie within +/- 2 standard errors[1] of the estimate. Therefore, in the foregoing query given the low standard error computation, there is a high probability that the estimated sums are accurate to within about a 2 percent error.

Attention: When the original table is very large and the sampling rate is not extremely small, we can typically be more specific about the precision of our estimator. For example, the true value of the sum is within +/- 1.96 standard errors with probability approximately 95%, and within +/- 2.576 standard errors with probability approximately 99%.

The optimizer can treat the sampling predicate like any other predicate for optimization purposes.

Important: The main drawback to this approach is that a scan of the entire table is required, so that there is no I/O savings. In practice, it may be desirable to amortize the sampling cost over multiple queries by saving the sample as a table. The sample should be refreshed periodically however, so that sampling anomalies do not systematically influence the results.

[1] Sampling Techniques by Cochran ISBN 0-471-16240-X

In the following queries, we obtain a better estimate of total sales for each group, by scaling up using the true sampling rate, that is, the group size in the entire table divided by the group size in the sampled table. This scaleup, though more expensive to compute, leads to more stable and precise estimators.

The SQL shown in Example 4-2 creates the sample table.

Example 4-2 Create sample table

```
CREATE TABLE samp_table(country, year, amount) AS
    SELECT loc.country, YEAR(t.pdate), ti.amount
    FROM trans t, transitem ti, loc loc
    WHERE t.transid = ti.transid AND loc.locid = t.locid
        AND RAND(1) < :samp_rate
```

The SQL shown in Example 4-3 creates a view that computes the group size ‘g_size’.

Example 4-3 Compute the group size

```
CREATE TABLE big_group_sizes(country, year, g_size) AS
    SELECT loc.country, YEAR(t.pdate), COUNT(*)
    FROM trans t, transitem ti, loc loc
    WHERE t.transid = ti.transid AND loc.locid = t.locid
    GROUP BY loc.country, YEAR(t.pdate)
```

Important: You need to ensure that the appropriate indexes are created and statistics collected before running the following query. This is business as usual performance tuning activity. Typically indexes are created on join columns and grouping columns to improve the performance of queries. In the foregoing example, indexes could be created (if they do not already exist for semantic reasons) on columns trans.transid, transitem.transid, trans.locid, loc.locid, and loc.country.

The SQL shown in Example 4-4 scales up the estimate by the true sampling rate as highlighted.

Example 4-4 Scale the estimate by the true sampling rate

```
SELECT s.country, s.year, b.g_size * AVG(s.sales) AS est_sales,
 SQRT(b.g_size * b.g_size * ((1e0 - :samp_rate)/COUNT(s.amount))
  * (1e0 - (1e0/COUNT(s.amount))) * (COUNT(s.amount)/(COUNT(s.amount)-1e0))
  * VAR(s.amount)) AS std_err
FROM samp_table s, big_group_sizes b
WHERE s.country = b.country AND s.year = b.year
GROUP BY s.country, s.year, b.g_size
```

We do the scaleup by computing the average sales for a group in the sampled table (that is, total sales for the group divided by the group size), and then multiplying by g_size, the size of the group in the original table.

We also used a different standard error formula using the VAR function that corresponds to the different estimator.

4.2 Retail

We have selected the following typical business queries for our examples:

1. Present annual sales by region and city.
2. Provide total quarterly and cumulative sales revenues by year.
3. List the top 5 sales persons by region this year.
4. Compare and rank the sales results by state and country.
5. Determine relationships between product purchases.
6. Determine the most profitable items and where they are sold.
7. Identify store sales revenues noticeably different from average.
8. Project growth rates of Web hits for capacity planning purposes.

4.2.1 Present annual sales by region and city

This is a typical report reviewing sales results for planning budgets, campaigns, expansions/consolidations etc.

Data

Input for this report is primarily transaction data along with dimension information relating to date, product, and location. Attributes of interest include:

- Date of transaction, product purchased, product price and quantity purchase
- Product code, product name, subgroup code, subgroup name and product group and product group name
- Region, city

BI functions showcased

GROUP BY, ROLLUP

Steps

Our data resided in a FACT_TABLE and a LOOKUP_MARKET table.

The SQL shown in Example 4-5 was run in DB2 Control Center.

Example 4-5 Annual sales by region and city

```
SELECT b.region_type_id,  a.city_id, SUM(a.sales) AS TOTAL_SALES
FROM fact_table a, lookup_market b
WHERE YEAR(transdate)=1999 AND a.city_id=b.city_id
   AND b.region_type_id=6
GROUP BY ROLLUP(b.region_type_id,a.city_id)
ORDER BY  b.region_type_id, a.city_id
```

Note: To reduce the size of the query result, the foregoing SQL limits the query to region 6, and the transaction date to 1999.

Figure 4-1 shows the results of this query.

REGION_TYPE_ID	CITY_ID	TOTAL_SALES
6	1	81655
6	2	131512
6	3	58384
...	...	...
...	...	...
...	...	...
6	19	77113
6	20	55520
6	21	63647
6	22	7166
6	23	92230
...	...	...
...	...	...
6	30	1733
6	31	5058
6		1190902
		1190902

Figure 4-1 Yearly sales by city, region

Note: Some rows of the result table were removed to fit on the page. The result shows ROLLUP of two groupings (region, city) returning three totals as follows:

- Total for region, city
- Total for region
- Grand total

4.2.2 Provide total quarterly and cumulative sales revenues by year

This is again a typical report reviewing sales results for planning purposes. Our query reported on years 1993 to 1995.

Data

The main source of input for this query is transaction data with the key attributes of transaction date and transaction amount. Our data resides in two tables.

- TRANS contains dated transactions of all transaction records within a store. Each transaction contains a set of transaction items.
- TRANSITEM contains the associations with products and product groups.

BI functions showcased

OVER, PARTITION BY, ORDER BY

Steps

We executed the SQL shown in Example 4-6 via the DB2 Control Center:

Example 4-6 Sales revenue per quarter & cumulative sales over multiple years

```
SELECT YEAR(pdate) as year,
   QUARTER(pdate) as quarter,
   SUM(ti.amount) as quarter_sales,
   SUM(SUM(ti.amount)) OVER (PARTITION BY YEAR(pdate) ORDER BY
      QUARTER(pdate)) as cume_sales_year,
   SUM(SUM(ti.amount)) OVER (ORDER BY YEAR(pdate), QUARTER(pdate)) as
      cume_sales
FROM trans t, transitem ti
WHERE t.transid=ti.transid and year(pdate) BETWEEN 1993 AND 1995
GROUP BY YEAR(pdate),QUARTER(pdate)
```

Figure 4-2 shows the results of this query.

YEAR	QUARTER	QUARTER_SALES	CUME_SALES_YEAR	CUME_SALES
1993	1	1174175.54	1174175.54	1174175.54
1993	2	1319395.42	2493570.96	2493570.96
1993	3	966301.99	3459872.95	3459872.95
1993	4	1288172.59	4748045.54	4748045.54
1994	1	1359409.24	1359409.24	6107454.78
1994	2	1195574.30	2554983.54	7303029.08
1994	3	982637.52	3537621.06	8285666.60
1994	4	1092424.49	4630045.55	9378091.09
1995	1	1331141.25	1331141.25	10709232.34
1995	2	1145774.82	2476916.07	11855007.16
1995	3	1311809.02	3788725.09	13166816.18
1995	4	1180066.36	4968791.45	14346882.54

Figure 4-2 Cumulative sales by quarter, annually and reporting period

We visualized this result as a bar chart, as follows:

1. Using the DB2 command center, we saved our data into a text file.
2. We created the MS-Excel file from it, and then imported it into the Brio tool for creating charts.
3. With drag-and-drop of CUME_SALES values for years 1993, 1994, and 1995, from Figure 4-2, we created the charts shown in Figure 4-3 and Figure 4-4.

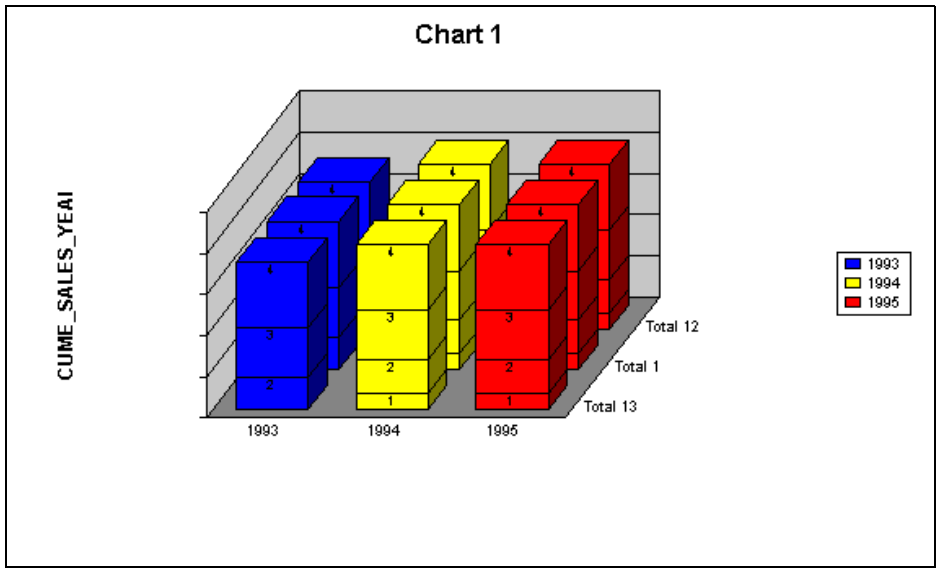


Figure 4-3 Cumulative sales by quarter and annually

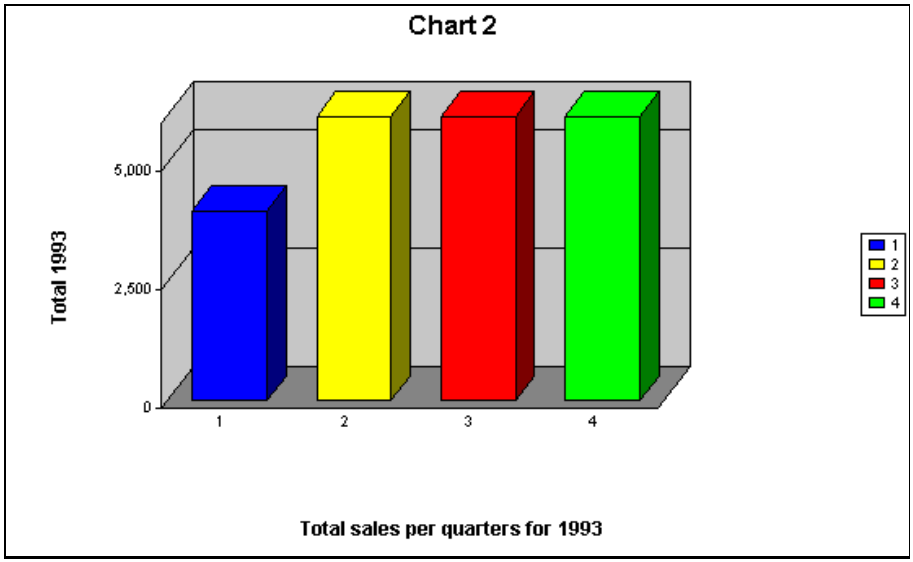


Figure 4-4 Cumulative sales by quarter for 1993

4.2.3 List the top 5 sales persons by region this year

This query requires the sales persons to have completed at least 10 sales transactions, and would typically be used for recognition purposes.

Data description

The main source of input to this query is sales information with the key attributes of date of sale, sales person, region, and count of sales transactions. All our data resides in the SALES table.

BI functions showcased

RANK, OVER, PARTITION BY, ORDER BY

Steps

We executed the SQL shown in Example 4-7 via the DB2 Control Center:

Example 4-7 Top 5 sales persons by region this year

```
WITH temp(region,sales_person,total_sale,sales_rank) AS
   (
   SELECT region, sales_person, COUNT(sales) AS total_sale,
      RANK() OVER (PARTITION BY region ORDER BY COUNT(sales) DESC) AS
      sales_rank
   FROM sales
   GROUP BY region, sales_person
   )
SELECT * FROM temp WHERE sales_rank <=5 AND total_sale >10
```

TOTAL_SALE counts the number of sales transactions.

Figure 4-5 shows the results of this query. Using a common table expression, a “virtual” table called ***temp*** is first created with results from number of sales (TOTAL_SALE) with partitioning over a region.The table ***temp*** *i*s then ranked to show the top five salesmen whose TOTAL_SALE is >=10.

REGION	SALES_PERSON	TOTAL_SALE	SALES_RANK
Manitoba	LEE	16	1
Manitoba	CHANG	14	2
Manitoba	GOUNOT	14	2
Manitoba	LUCCHESSI	12	4
Manitoba	ADAMS	11	5
Ontario-North	LUCCHESSI	16	1
Ontario-North	CHANG	15	2
Ontario-North	ADAMS	14	3
Ontario-North	LEE	14	3
Ontario-North	GOUNOT	12	5

Figure 4-5 Top 5 sales persons by region

4.2.4 Compare and rank the sales results by state and country

This query compares the sales results rolled up to country and state level for 1994. Its purpose is to view global sales ranking, peer to peer ranking among states and countries, with ranking within parent levels.

Data

Our data is sourced from sales transactions, using the same tables as used earlier. The key attributes are transaction date, transaction amount, state, and country. Our data resides in the following tables:

- TRANS
- TRANSITEM
- LOC

BI functions showcased

GROUPING, RANK, OVER, ORDER BY, ROLLUP

Steps

We executed multiple queries, each addressing a particular requirement.

Query 1

The query shown in Example 4-8 globally ranks the countries and states by the sales revenues:

Example 4-8 Globally rank the countries & states by sales revenues

```
SELECT SUM(ti.amount) AS sum, loc.country, loc.state,
   GROUPING(loc.country) + GROUPING(loc.state) AS level,
   RANK() OVER (ORDER BY SUM(ti.amount) DESC ) AS global_rank
FROM trans t, transitem ti, loc loc
WHERE t.transid =ti.transid AND loc.locid = t.locid AND YEAR(pdate) = 1994
GROUP BY ROLLUP (loc.country, loc.state)
ORDER BY global_rank
```

Figure 4-6 shows the results of this query.

SUM	COUNTRY	STATE	LEVEL	GLOBAL_RANK
4630045.55			2	1
1473235.28	USA		1	2
1030446.32	Germany		1	3
946140.59	Canada		1	4
715739.19	Australia		1	5
581233.54	Germany	BC	0	6
505079.27	Australia	BC	0	7
464484.17	UK		1	8
367717.45	USA	FL	0	9
274012.61	USA	CT	0	10
256508.73	Canada	MB	0	11
214101.47	Canada	NF	0	12
188043.92	Germany	CD	0	13
166649.88	USA	HI	0	14
163467.68	UK	CD	0	15
155686.91	UK	BC	0	16
150941.33	Germany	AB	0	17

Figure 4-6 Global ranking

States, countries, and the world have a level by hierarchy. The world has level 2, states have level 1, and the countries have a level 0 within the LEVEL as shown in Figure 4-7. The world is ranked 1 (GLOBAL_RANK) by virtue of the total sales transactions.

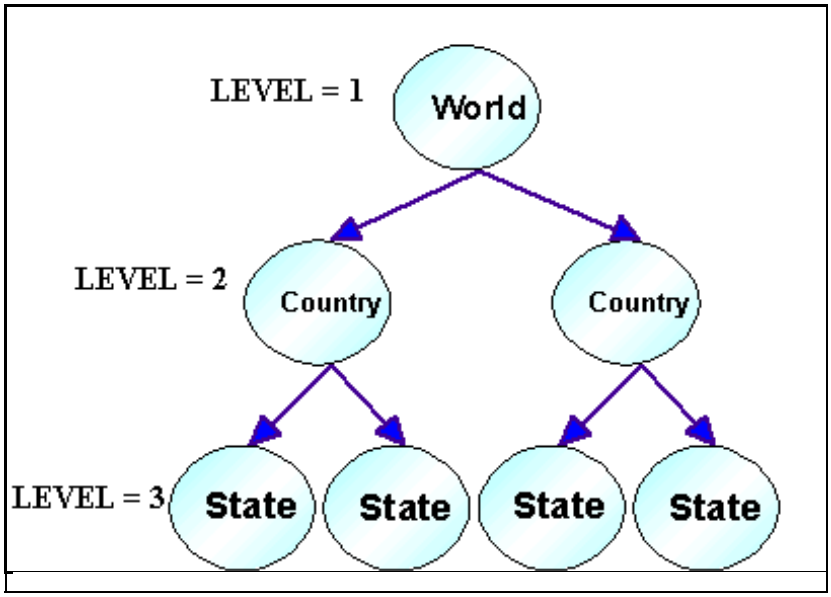


Figure 4-7 Levels in hierarchy

Query 2

The query shown in Example 4-9 ranks the sales among peers, that is, rank all the countries, and then the states, across multiple countries.

Example 4-9 Sales among peers

```
SELECT SUM(ti.amount) AS sum, loc.country, loc.state,
   GROUPING(loc.country) + GROUPING(loc.state) AS level,
   RANK () OVER (PARITION BY GROUPING (loc.country) + GROUPING(loc.state)
   ORDER BY SUM(ti.amount) DESC) AS rank_within_peers
FROM trans t, transitem ti, loc loc
WHERE t.transid = ti.transid AND loc.locid = t.locid AND YEAR(pdate) = 1994
GROUP BY ROLLUP (loc.country, loc.state)
ORDER BY level DESC, rank_within_peers
```

Figure 4-8 shows the results of this query.

SUM	COUNTRY	STATE	LEVEL	RANK_WITHIN_PEERS
4630045.55		-	2	1
1473235.28	USA	-	1	1
1030446.32	Germany	-	1	2
946140.59	Canada	-	1	3
715739.19	Australia	-	1	4
464484.17	UK	-	1	5
581233.54	Germany	BC	0	1
505079.27	Australia	BC	0	2
367717.45	USA	FL	0	3
274012.61	USA	CT	0	4
256508.73	Canada	MB	0	5
214101.47	Canada	NF	0	6
188043.92	Germany	CD	0	7
166649.88	USA	HI	0	8
163467.68	UK	CD	0	9
155686.91	UK	BC	0	10
150941.33	Germany	AB	0	11
150779.89	Canada	BC	0	12
145234.34	USA	GA	0	13
142927.50	Canada	NS	0	14
132767.59	USA	AZ	0	15
127419.56	Australia	CD	0	16
99841.51	USA	AR	0	17
86094.43	Germany	EF	0	18

Figure 4-8 Ranking within peers

Query 3

The query shown in Example 4-10 ranks the sales within each parent, that is, rank all the countries, and then states within each country.

Example 4-10 Sales within each parent

```
SELECT SUM(ti.amount) AS sum, loc.country, loc.state,
   GROUPING (loc.country) + GROUPING(loc.state) AS level,
   RANK() OVER (PARTITION BY GROUPING (loc.country) + GROUPING(loc.state),
   CASE WHEN GROUPING(loc.state) = 0 THEN loc.country END
   ORDER BY SUM (ti.amount) DESC) AS rank_within_parent
FROM trans t, transitem ti, loc loc
WHERE t.transid = ti.transid AND loc.locid = t.locid
   AND YEAR(pdate) = 1994
```

```
GROUP BY ROLLUP(loc.country, loc.state)
ORDER BY level desc,
CASE WHEN level =0 THEN loc.country END,
rank_within_parent
```

Figure 4-9 shows the results of this query.

SUM	COUNTRY	STATE	LEVEL	RANK_WITHIN_PARENT
4630045.55			2	1
1473235.28	USA		1	1
1030446.32	Germany		1	2
946140.59	Canada		1	3
715739.19	Australia		1	4
464484.17	UK		1	5
505079.27	Australia	BC	0	1
127419.56	Australia	CD	0	2
83240.36	Australia	AB	0	3
256508.73	Canada	MB	0	1
214101.47	Canada	NF	0	2
150779.89	Canada	BC	0	3
142927.50	Canada	NS	0	4
62451.81	Canada	ON	0	5
55162.44	Canada	AB	0	6
50802.29	Canada	NT	0	7
13406.46	Canada	NB	0	8
581233.54	Germany	BC	0	1

Figure 4-9 Ranking within parent

4.2.5 Determine relationships between product purchases

The purpose of this query is to try and establish whether there is a relationship between products purchased by customers, so that it can be used by sales persons to cross-sell complementary products.

It is well known that there is a strong relationship between certain product purchases, such as hammers and nails, paint and paint brushes and sanding paper, etc.

However, other relationships may not be so readily apparent. For example, a supermarket chain discovered a relationship between beer and candies (sweets), while another retailer discovered a relationship between late-night gasoline purchases and flowers.

Data mining is often used to discover unexpected or complex relationships; however, it is possible to use DB2 UDB's CORRELATION function to identify the nature of a relationship between 2 sets of data.

Many retailers now offer "LOYALTY" cards with the intention of being able to collect data based on peoples purchase pattern and thereby create targeted sales campaigns. Often these campaigns are based on very simple analysis of large volumes of data.

Data

The main source of data is the transactions obtained from purchases from a loyalty card scheme database. The key attributes of interest in our example are card number and the purchases of six items (coffee, beer, snack foods, bread, ready meals and milk).

BI functions showcased

CORRELATION

Steps

The steps we followed are shown in Example 4-11.

Example 4-11 Relationship between product purchases

```
SELECT DEC(CORRELATION(beer,snacks),15,2) AS "Beer_Snacks",
   DEC(CORRELATION(beer,milk),15,2) as "Beer_Milk"
FROM lc_purchases
```

The SQL in this example does two simple correlation calculations between purchases (dollar amount) of beer and snack foods, and beer and milk. Figure 4-10 shows the result of this query.

Beer_Snacks	Beer_Milk
0.83	0.01

Figure 4-10 CORRELATION output

The sample data shows a very high correlation between purchases (dollar amount) of beer and snack foods, but almost no correlation between beer and milk.

The sample data used in the foregoing example was charted using BRIO. Figure 4-11 and Figure 4-12 show that in our given sample, almost everyone who bought beer also bought some snack foods. However, only one person bought beer and milk.

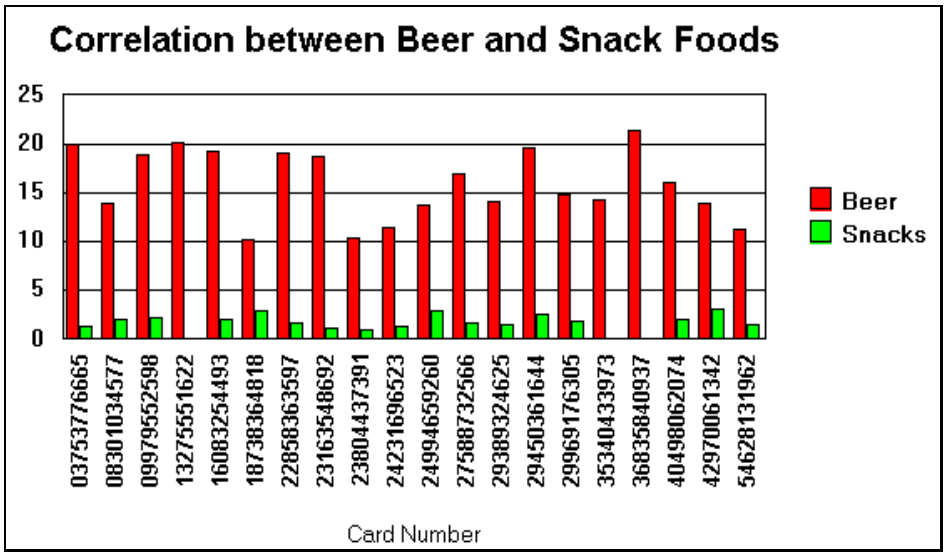


Figure 4-11 Correlation of purchases of beer and snack foods

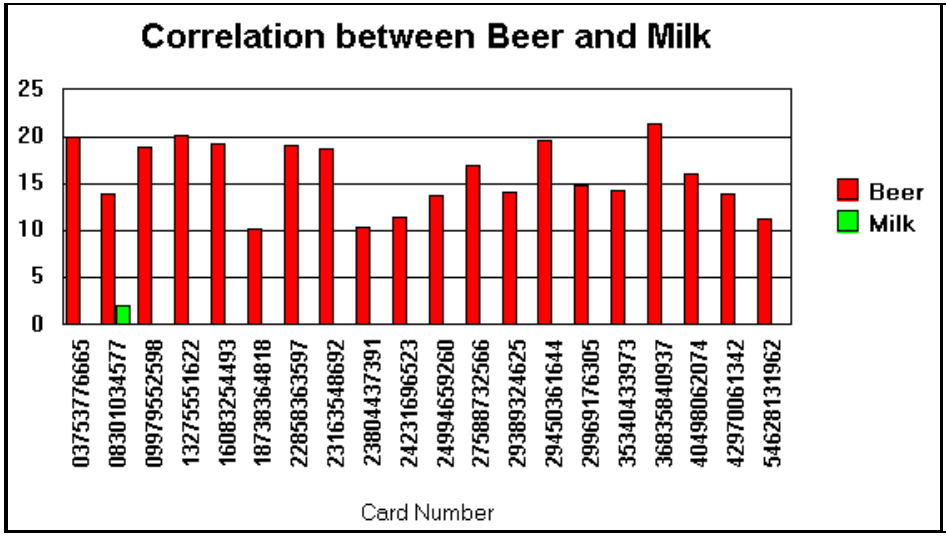


Figure 4-12 Correlation of purchases of beer and milk

4.2.6 Determine the most profitable items and where they are sold

Being able to determine one's most profitable items and where they are sold enables an organization to optimize product purchases and product distribution for maximum profitability.

To establish the profit on an item you need to know its cost to you and the price at which it is sold. In this example we will use the sales figures from a fictional world-wide Coffee Retailer and cross calculate the profit based on the profit of these items in their respective countries.

Data

Our data is mainly taken from pricing and transactions.

- The PRICING view includes store, item, and sales information.
- The TRANSACTIONS view includes store, item, cost, and price sold.

BI functions showcased

RANK, OVER, PARTITION BY, ORDER BY

Steps

This query is answered via multiple steps as follows:

1. For each variety of coffee, determine the store with the highest profit.
2. For each store, determine the coffee variety with the highest profit.
3. Determine the most profitable product in each store.
4. Determine the most profitable store for each variety of coffee.

Step 1

The query shown in Example 4-12 calculates the store with the highest profit on the different varieties of coffee:

Example 4-12 Store with the highest profit on the different varieties of coffee

```
SELECT store, item, profit
FROM
   (
   SELECT store, item, price - cost AS profit,
   RANK() OVER (PARTITION BY item ORDER BY price - cost DESC) AS
   rank_profit
   FROM pricing
   ) AS ranked_profit
WHERE rank_profit = 1
ORDER BY profit DESC
```

Figure 4-13 shows the results of this query.

STORE	ITEM	PROFIT
New York	Columbian	0.30
Tokyo	Java	0.25
New York	Mocha	0.25
New York	Kona	0.23

Figure 4-13 Store with highest profit of each variety of coffee

Step 2

The query shown in Example 4-13 determines the coffee variety delivering the highest profit in each store:

Example 4-13 Coffee variety delivering the highest profit in each store

```
SELECT store, item, profit
FROM
   (
   SELECT item, store, price - cost AS profit,
      RANK() OVER (PARTITION BY store ORDER BY price - cost DESC) AS
      rank_profit
   FROM pricing
   ) AS ranked_profit
WHERE rank_profit = 1
ORDER BY profit DESC
```

Figure 4-14 shows the results of this query.

STORE	ITEM	PROFIT
New York	Columbian	0.30
Tokyo	Java	0.25
Ankara	Columbian	0.19
Sydney	Kona	0.17
Berlin	Java	0.15

Figure 4-14 Highest profit of all varieties of coffee in a given store

In the foregoing examples it is obvious that New York has the highest profits for most of the varieties. However, this does not necessarily mean the most profits.

Step 3

The query shown in Example 4-14 calculates the most profitable product in each store.

Example 4-14 Most profitable product in each store

```
WITH tt AS
   (
   SELECT store, item, SUM(sales) AS total
   FROM transactions
   GROUP BY store, item
   )
SELECT store, item, total_profit
FROM
   (
   SELECT a.store,a.item,b.total*(a.price - a.cost) AS total_profit,
      RANK() OVER (PARTITION BY b.store ORDER BY b.total(a.price - a.cost)
      DESC) AS rank_profit
   FROM pricing a, tt b
   WHERE a.store=b.store AND a.item=b.item
   ) AS ranked_profit
WHERE rank_profit = 1
ORDER BY 3 DESC
```

Figure 4-15 shows the results of this query.

STORE	ITEM	TOTAL_PROFIT
Ankara	Columbian	75124.29
New York	Columbian	73518.90
Tokyo	Java	63117.25
Sydney	Kona	37451.00
Berlin	Java	36306.45

Figure 4-15 Most profitable product in each store

Step 4

The query shown in Example 4-15 calculates the most profitable store for each variety of coffee:

Example 4-15 Most profitable store for each variety of coffee

```
WITH tt AS
   (
   SELECT item, store, SUM(sales) AS total
   FROM transactions
   GROUP BY store, item
   )
SELECT item, store, total_profit
FROM
   (
```

```
    SELECT a.store,a.item,b.total*(a.price - a.cost) AS total_profit,
       RANK() OVER (PARTITION BY b.item ORDER BY b.total(a.price - a.cost)
       DESC) AS rank_profit
    FROM pricing a, tt b
    WHERE a.store=b.store AND a.item=b.item
    ) AS ranked_profit
WHERE rank_profit = 1
ORDER BY 3 DESC
```

Figure 4-16 shows the results of this query.

ITEM	STORE	TOTAL_PROFIT
Columbian	Ankara	75124.29
Kona	Ankara	68586.12
Java	Tokyo	63117.25
Mocha	Ankara	61966.72

Figure 4-16 Most profitable store for each variety of coffee

The data from these results is charted in Figure 4-17 and Figure 4-18.

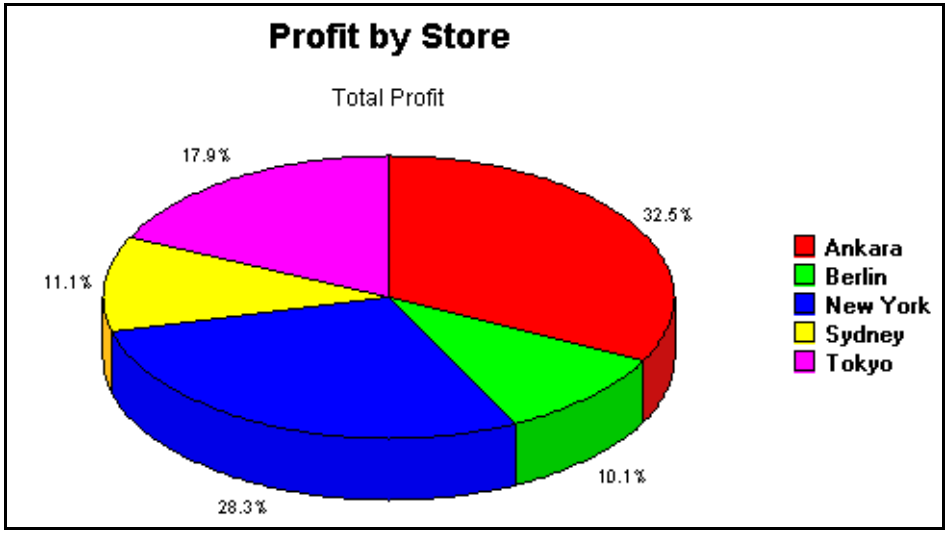


Figure 4-17 Total profit by store

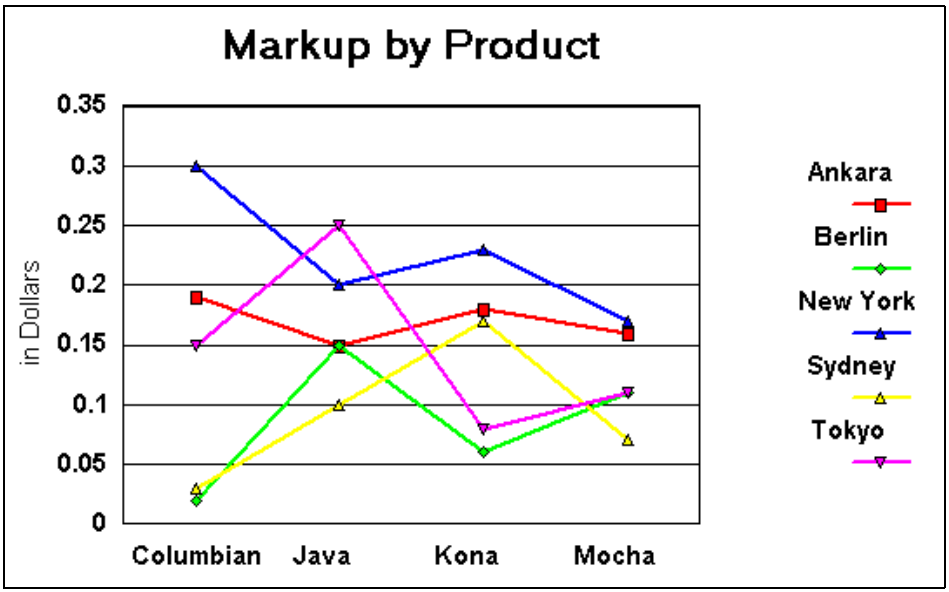


Figure 4-18 Profit by product in each store

These figures show that Ankara is below New York in profit for every product, while being the more profitable store.

4.2.7 Identify store sales revenues noticeably different from average

The purpose is to identify leading and lagging stores with the intention of either shutting down or initiating sales campaigns in poorly performing stores, or analyzing the better performing stores for their best practices to apply to other stores.

In this example we will use the sales figures from a fictional world-wide Coffee Retailer and calculate the revenues in their respective countries.

Data

The attributes in this example are the same pricing and transactions tables used in the previous example.

BI functions showcased

AVG, STDDEV

Steps

The steps we followed are shown in Example 4-16.

Example 4-16 Sales revenues of stores noticeably different from the mean

```
WITH t1(item,store,total) AS
   (
   SELECT item, store, SUM(sales)
   FROM transactions
   GROUP BY store, item
   ),
   t2(store,total_revenue) AS
      (
      SELECT a.store, SUM(b.total*a.price)
      FROM pricing a, t1 b
      WHERE a.store=b.store AND a.item.=b.item
      GROUP BY a.store
      ),
      t3 (avg_rev, std_rev) AS
         (
         SELECT AVG(total_revenue) AS avg_revenue,
         STDDEV(total_revenue) AS dev_revenue
         FROM t2
      )
SELECT a.store,a.total_revenue,
   (a.total_revenue - b.avg.rev)/b.std_rev AS deviations
FROM t2 a, t3 b
```

In the foregoing SQL, we create three “virtual” tables to arrive at the final result, as follows:

1. t1 is a summary of sales by store and item. Sales is expressed in terms of dollars.
2. t1 is then used in conjunction with the pricing table to create, t2, a table of revenues.
3. t3 is average and standard deviation of the data from t2.

Finally, we calculate how many standard deviations each stores’ mean is from the overall mean.

Figure 4-19 shows the results of this query.

STORE	TOTAL_REVENUE	DEVIATIONS
Ankara	693786.15	-1.008606315...
Berlin	718827.00	-0.929581358...
New York ...	1066861.95	0.168761817...
Sydney	1017059.40	0.011592857...
Tokyo	1570395.20	1.757832998...

Figure 4-19 Store revenue and deviation from mean

Figure 4-20 charts this relationship.

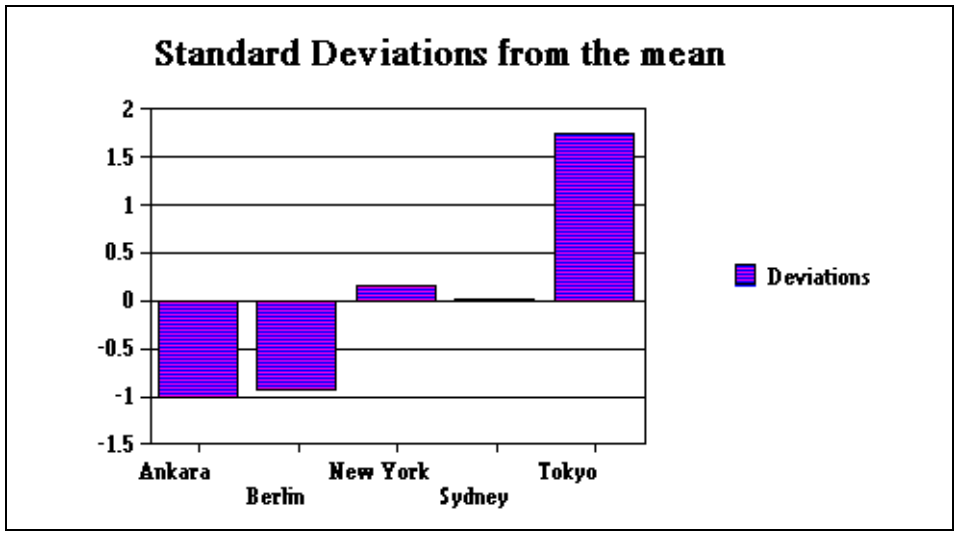


Figure 4-20 Standard deviations from the mean by revenue

From our previous example, and this one, Ankara makes almost twice the profit of the Tokyo store on almost half the revenue, thus making Ankara the most profitable store.

4.3 Finance

1. Identify the most profitable customers
2. Identify the profile of transactions concluded recently
3. Identify target groups for a campaign
4. Evaluate effectiveness of a marketing campaign
5. Identify potential fraud situations for investigation
6. Plot monthly stock prices movement with percentage change
7. Plot the average weekly stock price in September
8. Project growth rates of Web hits for capacity planning purposes
9. Relate sales revenues to advertising budget expenditures

4.3.1 Identify the most profitable customers

Determining customers that buy products/services that bring the most profit to the company can be used to build stronger customer relationships and increase profitability and customer satisfaction.

Data

We assume that there is a data warehouse containing information about customers and products/services they subscribe to. And we assume that we have already done an analysis of how much profit each product or service brings to the company per year.

The main sources of our data are:

- Product information which lists the products/services offered
- Customer details
- Products/Services purchased by customers

Customer profitability is the price of the product minus the cost the company incurs in providing the service, as shown in Figure 4-21.

Customer yearly profit is calculated:

Customer Total Profit (CTP) = (N_1 x P_1) + (N_2 x P_2) +...(N_n x P_n)

Where:

N = number of product/service a customer has per product type

P = profit for product type

Figure 4-21 Profit from a customer

BI functions showcased

RANK, DENSE_RANK, ROW_NUMBER, ORDER BY

Companies rank their customers based on their business rules. In this example, we show ranking using RANK, DENSE_RANK and ROWNUMBER in order to show the differences in the results of these functions.

Steps

The SQL shown in Example 4-17 provides the desired result.

Example 4-17 Most profitable customers

```
SELECT a.custid,SUM(c.profit) AS total_profit,
   RANK() OVER (ORDER BY SUM(c.profit) DESC) AS rank,
   DENSERANK() OVER(ORDER BY SUM(c.profit) DESC) AS denserank,
   ROW_NUMBER() OVER(ORDER BY SUM(c.profit) DESC) AS rownum
FROM cust a, prod_owned b, prod c
WHERE a.custid=b.custid AND b.prodid=c.prodid
GROUP BY a.custid
```

Figure 4-22 shows the results of this query.

CUSTID	TOTAL_PROFIT	RANK	DENSERANK	ROWNUM
22	215.00	1	1	1
28	210.00	2	2	2
36	210.00	2	2	3
37	200.00	4	3	4
23	110.00	5	4	5
26	110.00	5	4	6
30	110.00	5	4	7
31	110.00	5	4	8
32	110.00	5	4	9
34	110.00	5	4	10
40	110.00	5	4	11
20	105.00	12	5	12
25	105.00	12	5	13
27	105.00	12	5	14
29	105.00	12	5	15
38	105.00	12	5	16
11	100.00	17	6	17
13	100.00	17	6	18
14	100.00	17	6	19
15	100.00	17	6	20

Figure 4-22 Customer profitability ranking result

We ran the query in IBM QMF and saved the output in an MS-Excel spreadsheet and used MS-Excel charting to create the bar chart shown in Figure 4-23.

Figure 4-23 Customer profitability bar chart

4.3.2 Identify the profile of transactions concluded recently

We look at the profile of transactions from two perspectives as follows:

- **Query 1** identifies the transaction amount range which accounted for most of the companies transactions. For how many of our transactions are worth less 9,000 dollars. This information can be used to identify spending patterns for targeted sales campaigns.
- **Query 2** identifies the value of transactions by percentage of volumes. For example, what is value of our transactions at the 90th percentile? This can be used in more detailed analysis. For example, if the retailer knows that 90 percent of his sales are less than 20,000 dollars, then he can infer that stocking many items more $20,000 may not be cost-effective. However, he can identify products within the most common ranges.

Data

The major source for this analysis are the transactions. In our example, this data is contained in 2 tables.

- TRANS which is the master transaction table which holds summary details of completed transactions.
- TRANSITEM is the sales details table which holds the individual products sold and their value by transaction.

BI functions showcased

SUM, ROWNUMBER, OVER, ORDER BY

Query 1

The SQL shown in Example 4-18 can be used to generate the result set that can then be displayed via an equi-width histogram. The transactions are assigned to a range bucket based on the sales value of a transaction.

Example 4-18 Equi-width histogram query

```
WITH dt AS
   (
   SELECT t.transid, SUM(amount) AS trans_amt,
      CASE
         WHEN (SUM(amount) - 0)/((60000 - 0)/20) <= 0 THEN 0
         WHEN (SUM(amount) - 0)/((60000 - 0)/20) >= 19 THEN 19
         ELSE INT((SUM(amount) - 0)/((60000 - 0)/20))
      END AS bucket
   FROM trans t, transitem ti
   WHERE t.transid=ti.transid
   GROUP BY t.transid
   )
SELECT bucket, COUNT(bucket) AS height,
   (bucket + 1)*(60000 - 0)/20 AS max_amt
FROM dt
GROUP BY bucket
```

In this query, assuming a maximum transaction amount of $60,000 (based on domain knowledge of this application), we create twenty $3000 range buckets in the common table expression, and then count the number of transactions in each bucket range. The CASE expression is used to assign a bucket to a particular transaction, and the result of the common table expression is a table by transaction of the transaction amount and the bucket it belongs to. The query querying the result of the common table expression, then groups the rows by bucket, counts the number of transactions in each bucket, and lists the upper range of the bucket for these transactions.

Figure 4-24 shows the results of this query.

BUCKET	HEIGHT	MAX_AMT
0	449	3000
1	609	6000
2	833	9000
3	706	12000
4	524	15000
5	401	18000
6	312	21000
7	200	24000
8	126	27000
9	61	30000
10	43	33000
11	32	36000
12	25	39000
13	8	42000
14	3	45000
15	4	48000
19	1	60000

Figure 4-24 Equi-width histogram data

This data was copied into Lotus 1-2-3 to generate the equi-width histogram shown in Figure 4-25.

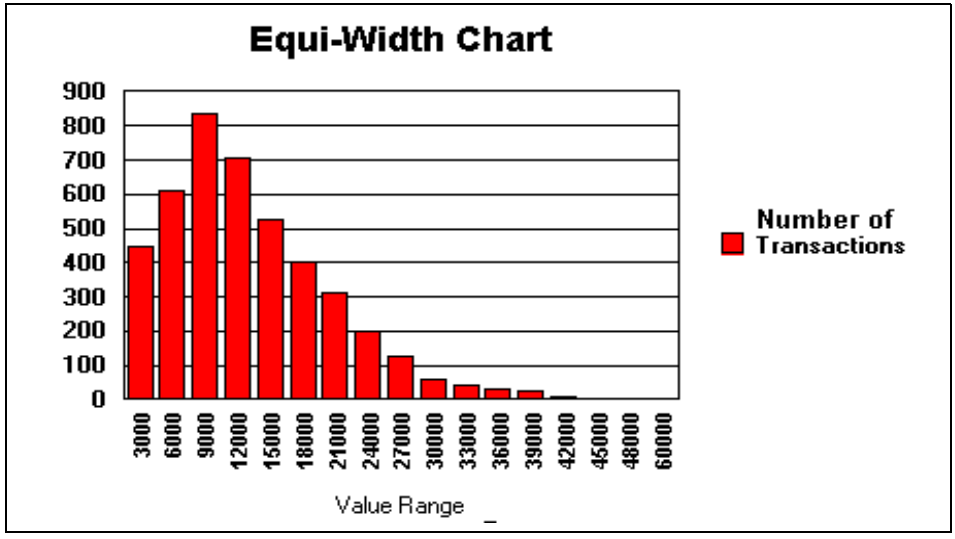


Figure 4-25 Equi-width chart

The data and histogram shows that a significant proportion of the transactions are less than fifteen thousand, with a peak in the six to nine thousand range. The answer to the number of transactions worth less than 6,000 dollars is 1058. The graph also shows that 449 transactions are less than 3000 dollars, and 609 transactions are between 3000 and 6000 dollars.

Query 2

The SQL shown in Example 4-19 can be used to generate the result set that can then be displayed via an equi-height histogram. The bucket boundaries are chosen so that each bucket contains approximately the same number of data points.

Example 4-19 Equi-height histogram query

```
WITH dt AS
   (
   SELECT t.transid, SUM(amount) AS trans_amt,
      ROWNUMBER() OVER (ORDER BY SUM(amount))*10/
         (
         SELECT COUNT(DISTINCT transid) + 1
         FROM transitem
         ) AS bucket
   FROM trans t, transitem ti
   WHERE t.transid=ti.transid
   GROUP BY t.transid
   )
SELECT (bucket+1)*10 AS percentile, COUNT(bucket) AS b_count,
   MAX(trans_amt) AS max_value
FROM dt
GROUP BY bucket
```

In this example we have used 10 buckets, so that 10% of the data points fall in each bucket. The internal bucket boundaries are often referred to as the 0.1, 0.2, ..., 0.9 quantiles of the data distribution, or as the 10th, 20th, ..., 90th percentiles. For example, Figure 4-26 shows that the 10th percentile for our data is $2840.05, that is, 10% of transactions have a dollar value less than this amount.

In effect, the query computes the total sales amount for each of ‘n’ transactions as trans-amt, sorts the amounts in increasing order, and assigns the number one to the smallest transaction, two to the next largest transaction, etc. using the ROWNUMBER() function. Multiplying these numbers by 10 (the number of buckets), and then dividing these numbers by ‘n’ (which is computed as COUNT(DISTINCT transid)), and rounding to the nearest integer produces the desired bucket number for each transaction.

Figure 4-26 shows the results of this query.

PERCENTILE	B_COUNT	MAX_VALUE
10	433	2840.05
20	434	5229.95
30	434	7059.54
40	434	8484.14
50	433	9802.23
60	434	12025.50
70	434	14527.10
80	434	17513.65
90	434	21926.91
100	433	60258.15

Figure 4-26 Equi-height histogram data

As mentioned earlier, that the MAX_VALUE represents the transaction amount at the bucket boundary.

This data is copied into Lotus 1-2-3 to create the equi-height histogram shown in Figure 4-27. The histogram will display the max values in each bucket as the width. It is called an “equi-height” histogram because the ranges are set to the same height. The data in the graph has been manipulated so that the max value is the boundary. The individual figures have had the preceding value subtracted.

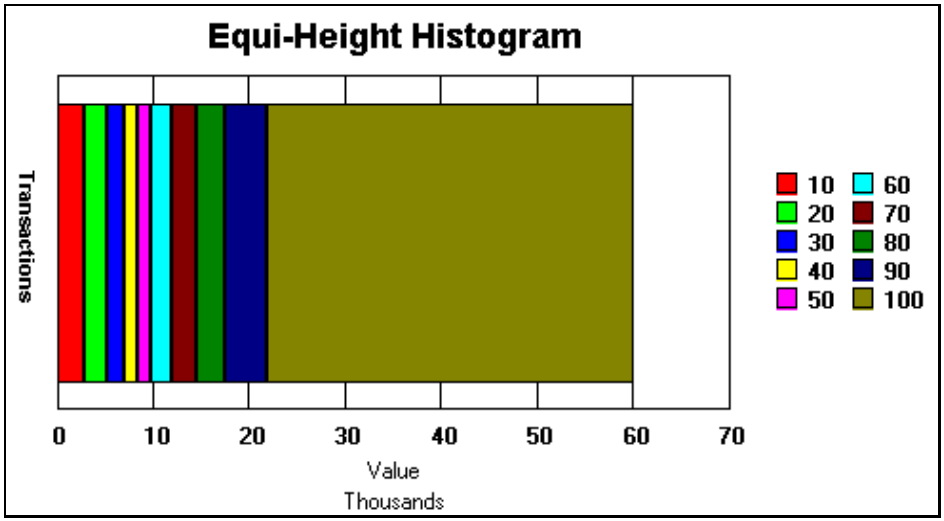


Figure 4-27 Equi-height histogram

The data and histogram shows that 30% of all transactions are worth less than 7,060 dollars.

4.3.3 Identify target groups for a campaign

The objective is to identify a particular group of customers from a larger set that will most likely respond to a marketing campaign, thus resulting in a better return on investment.

In our example, a financial institution would like to increase revenue by marketing mortgages to customers during the first quarter of the new fiscal year. Previous wider coverage state-wide and county-wide campaigns have been unsuccessful, and the company would like to focus on a particular city for better results.

Candidate target cities chosen are Palo Alto (8 branches) and San Jose (9 branches) since they are located in the Silicon Valley area which has highest average family income in the state of California. The rationale being that people with high incomes generally own their homes or would like to own homes, and therefore are ideal targets for the campaign.

The decision to choose Palo Alto or San Jose as the target city was based on the following analysis.

1. A survey was conducted of the residents of Palo Alto and San Jose and the results were analyzed.
2. Using the Chi-squared technique, infer from the results if a relationship exists between where a customer lives and the product (s)he will buy.
3. If such a relationship exists, use the Wilcoxon Rank Sum Test to prove that Palo Alto residents will likely buy mortgage loans.

Data

The main sources of data for this query are:

- Cumulative Distribution of the Chi-square table
- Survey results in the form of Survey tables
- Cumulative Distribution of the Wilcoxon Rank-Sum Statistic table

BI functions showcased

SUM, CUBE, LOG(n), GROUPING, RANK

Steps

We executed the following three steps to arrive at the answer.

Step 1

This involved collecting the survey data and loading it into a DB2 table called SURVEY. An SQL query (not shown here) was issued against the SURVEY table it to arrive at a Contingency table as shown in Table 4-1.

Table 4-1 Survey data contingency table

Product	Palo Alto	San Jose	Total
checking/savings	45	85	130
visa	30	10	40
mortgage	110	80	190
Total	**185**	**175**	**360**

This data was then loaded into a DB2 table SURVEY.

Step 2

We used the Chi-squared technique to prove that there is a statistically significant relationship between where customer lives and the product he/she will buy by disproving the null hypothesis. In other words, disproving that where customers live has no bearing on what products they buy.

The formula for the “maximum likelihood” Chi-squared test for independence is shown in Table 4-2.

Table 4-2 Chi-squared test for independence test statistic

Chi-squared Test for Independence Test Statistic:

Maximum Likelihood X = 2_n log ($_n$)
+ [2_n11 log (n11) + ... + 2n rc
- [2_n1+ log (n1+) + ... + 2n$_r$+ log (n$_r$+)]
- [2_n+1 log (n+1) + ... + 2n+c log $_{(n+c)}$]

Where:
nij: # in cell (i,j)
ni+: row i sum
n+j: column j sum
n: total # of user

Note: Table lookups and complex calculations can be automated in DB2 via User Defined Functions (UDFs).

The SQL query for calculation of the foregoing Chi-square (X) is shown in Example 4-20.

Example 4-20 Chi-square computation

```
WITH c_table (prod_name, city, n, g1, g2) AS
   (
      SELECT prod_name, city, count(*), 2e0*(0.5e0-GROUPING(prod_name)),
      2e0*(0.5e0-GROUPING(city))
   FROM survey
   GROUP BY CUBE (prod_name, city)
   )
SELECT  SUM(g1*g2*2e0*n*log(n)) as chi2
FROM c_table
```

Important: Consider using floating point data type instead of INTEGER or DECIMAL to avoid arithmetic exceptions such as overflow, when multiple divide and multiplication operators involved.

The result of this query is shown in Figure 4-28.

CHI2
34.1139

Figure 4-28 Chi-Squared value of city and product preference relationship

The Cumulative Distribution of the Chi-square table (not shown here) shows that under the null hypothesis, there is no statistical relationship between where the customer lives and the products (s)he will buy. The probability of seeing a Chi-square values of 34.1139 or higher is less than 0.001%.

Therefore, we conclude that there is a relationship between where a person lives and the products (s)he will buy.

Step 3

Here we use the Wilcoxon Rank Sum Test to prove that Palo Alto residents will likely buy more mortgages than the residents of San Jose.

Here too, it involves **disproving** the null hypothesis, that is we disprove that customers in Palo Alto will **not** buy more mortgages than the residents of San Jose. This involves the following steps:

a. Collect the percentage of people who responded “yes” to likely buying mortgages for each branch in Palo Alto and San Jose. Load the statistics to a table called SURVEY_MORTG. The columns in this table are:

   Cityid - city
   Branchid - branch within the city
   PercentYes - Percentage of branch customers who responded YES

b. Compute the Wilcoxon Rank Sum Test one-tail analysis for Palo Alto — also known as the 'W' statistic.

c. Look up the table of tail probabilities for this 'W' statistic value, in order to determine whether the null hypothesis is disproved or not.

Step 3b

Compute the 'W' statistic for Palo Alto. Consider Example 4-21.

Example 4-21 Compute the 'W' statistic

```
WITH ranked_mortg (city, ranks) AS
   (
   SELECT city, RANK() OVER (ORDER BY PercentYes)
   FROM survey_mortg
   )
SELECT SUM(ranks) as W
FROM ranked_mortg
WHERE city='Palo Alto'
```

The result of this query is shown in Figure 4-29.

W
93

Figure 4-29 Wilcoxon W

Step 3c

Based on the Table of Tail probabilities for small values of ranks (not shown here), the probability of getting a rank sum of 93 is only 2.3% under the null hypothesis, which is quite low. We therefore conclude that Palo Alto residents will likely respond to our mortgage loan campaign a compared to San Jose residents.

Attention: Table lookups and complex calculations can be automated in DB2 via User Defined Functions (UDFs).

4.3.4 Evaluate effectiveness of a marketing campaign

The objective is to evaluate the effectiveness of a mortgage loans campaign, so that a decision can be made whether to expand it to other areas, or change tactics altogether.

The scenario is that a month-long (February) marketing campaign in Palo Alto just ended. The campaign sold mortgage loans to homeowner customers who do not have mortgages with the company.

The effectiveness of a campaign is determined by comparing the results of the target city where the campaign was run, with a comparable “control” city where no campaign was done. San Jose was selected as the “control” city of this campaign since it is comparable to Palo Alto in the number of branches. San Jose has nine (9) branches and Palo Alto has eight (8) branches.

Data

The main data source is an existing data warehouse containing the following key attributes:

- Customer identification
- Product identification
- City (location of branch)
- Branch identification
- Customer income range
- Number of mortgage loans acquired
- Date of transaction (of mortgage loan)

We queried the data warehouse to accumulate February mortgage loan sales for each branch in San Jose and Palo Alto, and then inserted the query results into a “feb_sales” table.

The Cumulative Distribution of the Wilcoxon Rank-Sum Statistic table is used here as well.

BI functions showcased

RANK, DENSE_RANK, ROW_NUMBER, ORDER BY, CORRELATION, CUBE, SUM

Steps

We executed the following three steps to answer our query:

- Load the data from a data warehouse into a “feb_sales” table
- Determine whether the campaign was successful using the Wilcoxon Rank Sum Test
- Report the results

Step 1

This involves loading the data from the data warehouse, and loading it into a DB2 “feb_sales” table.

Step 1a

The SQL shown in Example 4-22 was executed in DB2 Control Center to generate data for our feb_sales table which was located on a different DB2 server.

Example 4-22 Generate feb_sales data

```
SELECT city, branch_id, sum(qty) AS sales
FROM sales_dtl
WHERE (city='San Jose' OR city='Palo Alto') AND YEAR(date_sold)=2001
   AND MONTH(date_sold)=02
GROUP BY city,branch_id
```

Step 1b

The results of this query was saved in a comma delimited (DEL) file and transported to the target DB2 server.

Step 1c

The DEL file was imported into the “feb_sales” table on the target DB2 server.

Step 2

This involves using the Wilcoxon Rank Sum Test to prove that the campaign was successful by disproving the null hypothesis

a. Compute the Wilcoxon Rank Sum Test one-tail analysis for Palo Alto — also known as the ‘W’ statistic

b. Look up the table of tail probabilities for this ‘W’ statistic value, in order to determine whether the null hypothesis is disproved or not

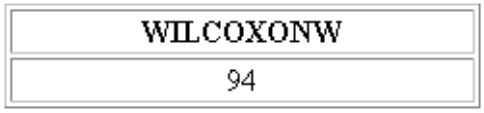
Step 2a

Compute the ‘W’ statistic for Palo Alto. Consider Example 4-23.

Example 4-23 Compute the ‘W’ statistic

```
WITH ranked_sales (city, ranks) AS
   (
   SELECT city, RANK() OVER (ORDER BY sales)
   FROM feb_sales
   )
SELECT SUM(ranks) as W
FROM ranked_sales
WHERE city='Palo Alto'
```

The result of this query is shown in Figure 4-30.



WILCOXONW
94

Figure 4-30 Wilcoxon W

Step 2b

Based on the Table of Tail probabilities for small values of ranks (not shown here), the probability of getting a rank sum of 94 is only 2.3%. In other words, the probability that Palo Alto campaign was not successful is about 2.3%, which is quite low. We therefore conclude the campaign was successful.

Attention: Table lookups and complex calculations can be automated in DB2 via User Defined Functions (UDFs).

Step 3

Once the campaign is considered successful, the following additional information can be gathered.

a. Who are the customers that responded positively to the mortgage loan campaign? List top ten Palo Alto customers who got their mortgages during February.

b. Was there a relationship between income and mortgage loans?

   For each customer, count the mortgages he/she has and determine its relationship to customer's income range using DB2 CORRELATION function (assuming a linear relationship).

c. Report Palo Alto Total Sales.

d. Report Palo Alto Branches' sales by city, branch, month (one row for each month).

e. Report Palo Alto Branch 1 sales by city, branch, month (one row for product).

Step 3a

List top 10 Palo Alto customers who got mortgages in February. Consider Example 4-24.

Example 4-24 Top 10 Palo Alto customers who got mortgages in February

```
WITH rank_tab (custid, mortg_count, rank, denserank, rownumber) AS
   (
   SELECT a.custid, count(b.prod_type) AS mortg_count,
      RANK () OVER (ORDER BY count(b.prod_type))  AS rank,
      DENSE_RANK () OVER (ORDER BY COUNT(b.prod_type)) AS denserank,
      ROW_NUMBER () OVER (ORDER BY COUNT(b.prod_type)) AS rownumber
   FROM cust a, prod_owned b
   WHERE a.custid=b.custid  AND city = 'Palo Alto' AND
   MONTH(b.open_date)=02 AND PROD_TYPE=3
   GROUP BY a.custid
   )
SELECT custid, mortg_count, rank, denserank, rownumber
FROM rank_tab
WHERE rownumber <=10
```

The results of this query are shown in Figure 4-31.

CUSTID	MORTG_COUNT	RANK	DENSERANK	ROWNUMBER
11	2	1	1	1
13	2	1	1	2
14	2	1	1	3
15	2	1	1	4
18	2	1	1	5
20	2	1	1	6
23	2	1	1	7
25	2	1	1	8
26	2	1	1	9
27	2	1	1	10

Figure 4-31 Top Ten Palo Alto customers who got mortgages in February

Step 3b

Are income and mortgage related? Consider Example 4-25.

Example 4-25 Are income and mortgage related?

```
WITH cust_tab (cusitd, income_range, total_mortg_loans)AS
   (
   SELECT a.custid, a.income_range,
      COUNT(b.prod_type) AS total_mortg_loans
   FROM cust a, prod_owned b
   WHERE a.custid=b.custid  AND MONTH(b.open_date)=02 AND PROD_TYPE=3
   GROUP BY a.custid, income_range, b. prod_type
   )
SELECT CORR( income_range, total_mortg_loans) from cust_tab
```

The results of this query are shown in Figure 4-32.

CORRELATION_INCOME_MORTG_LOANS
-0.7545929350231552

Figure 4-32 Negative correlation between income range and mortgage loans

The result is -0.7545929350231552, indicating that income range and mortgage loans may be inversely related. The statistical significance of this number is probably not totally intuitive and may not be true in all demographic areas. We therefore need to test further for statistical significance.

Step 3c

Report Palo Alto total sales.

We limited the query to Branch 1 to ensure result fits just a page. Consider Example 4-26.

Example 4-26 Palo Alto total sales

```
SELECT a.city, a.branch_id, b.prod_name, COUNT(*)  AS products_sold
FROM prod_owned a, prod b
WHERE a.prodid=b.prodid AND a.city='Palo Alto' AND a.branch_id=1
GROUP BY CUBE(a.city, a.branch_id, b.prod_name)
ORDER BY a.city, a.branch_id, b.prod_name
```

The results of this query are shown in Figure 4-33.

CITY	BRANCH_ID	PROD_NAME	PRODUCTS_SOLD
Palo Alto	1	MORTGAGE	46
Palo Alto	1	SAV/CHECK	18
Palo Alto	1	VISA	12
Palo Alto	1		76
Palo Alto		MORTGAGE	46
Palo Alto		SAV/CHECK	18
Palo Alto		VISA	12
Palo Alto			76
	1	MORTGAGE	46
	1	SAV/CHECK	18
	1	VISA	12
	1		76
		MORTGAGE	46
		SAV/CHECK	18
		VISA	12
			76

Figure 4-33 Palo Alto branch 1 total sales by product

Step 3d

Report Palo Alto Branches' Total Monthly sales, one row per month. Consider Example 4-27.

Example 4-27 Total monthly sales

```
SELECT a.city, a.branch_id, COUNT(*)  AS month_prod_sold,
   MONTH(open_date) AS month
FROM prod_owned a, prod b
WHERE a.prodid=b.prodid AND a.city='Palo Alto'
GROUP BY a.city, a.branch_id, MONTH(open_date)
```

The results of this query are shown in Figure 4-34.

CITY	BRANCH_ID	MONTH_PROD_SOLD	MONTH
Palo Alto	1	12	1
Palo Alto	1	64	2
Palo Alto	2	12	1
Palo Alto	2	64	2
Palo Alto	3	12	1
Palo Alto	3	62	2
Palo Alto	4	12	1
Palo Alto	4	62	2
Palo Alto	5	12	1
Palo Alto	5	62	2
Palo Alto	6	12	1
Palo Alto	6	62	2
Palo Alto	7	11	1
Palo Alto	7	57	2
Palo Alto	8	11	1
Palo Alto	8	59	2
Palo Alto	9	11	1
Palo Alto	9	59	2

Figure 4-34 Palo Alto branches' total monthly sales, one row per month

Step 3e

Report Palo Alto Branches' Total Monthly sales. Consider Example 4-28.

Example 4-28 Palo Alto branches' total monthly sales

```
SELECT city, branch_id, m1_sales, m2_sales
FROM
   (
   SELECT a.city, a.branch_id, MONTH(open_date) as month,
      COUNT(*) AS m1_sales,
      SUM(COUNT(*)) OVER(PARTITION BY a.city, a.branch_id ORDER BY
      MONTH(open_date)
      ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING) AS m2_sales
   FROM prod_owned a, prod b
   WHERE a.prodid=b.prodid AND a.city='Palo Alto'
   GROUP BY a.city, a.branch_id, MONTH(open_date)
   ) as dt
WHERE MONTH=1
```

The results of this query are shown in Figure 4-35.

CITY	BRANCH_ID	M1_SALES	M2_SALES
Palo Alto	1	12	64
Palo Alto	2	12	64
Palo Alto	3	12	62
Palo Alto	4	12	62
Palo Alto	5	12	62
Palo Alto	6	12	62
Palo Alto	7	11	57
Palo Alto	8	11	59
Palo Alto	9	11	59

Figure 4-35 Palo Alto branches' total monthly sales

4.3.5 Identify potential fraud situations for investigation

The objective is to determine potential credit card fraud by looking for large variations in current purchases as compared to past purchase practices.

Data

- A Profile view maintains the average and standard deviation of each customer's credit card purchases over a reference period.
- A Big_charges table that has customer transactions whose charge card amounts that are more than two standard deviations above the average.

BI functions showcased

AVG, **STDDEV**

Steps

1. First create a customer card usage PROFILE view.
2. Detect unusually large charges and insert a row into the BIG_CHARGES table.
3. Assume there is a trigger on the BIG_CHARGES table that sets off an alarm if for example, a customer has more than three big charges within a 12 hour period. This is not shown here.

Step 1

Create a customer usage PROFILE view as shown in Example 4-29.

Example 4-29 Customer usage profile view

```
CREATE VIEW profile(cust_id, avg_amt, sd_amt) AS
   SELECT cust_id, AVG(charge_amt), STDDEV(charge_amt)
   FROM custtrans
   WHERE date BETWEEN '2002-01-01' and '2002-03-31'
   GROUP BY cust_id
```

Step 2

Detect and flag unusually large charges by writing them to the BIG_CHARGES table as shown in Example 4-30.

Example 4-30 Detect & flag unusually large charges

```
CREATE TRIGGER big_chrg
AFTER INSERT ON custtrans
REFERENCING NEW AS newrow FOR EACH ROW MODE DB2SQL
WHEN (newrow.charge_amt > (SELECT avg_amt + 2e0 * sd_amt
                           FROM profile
                           WHERE profile.cust_id = newrow.cust_id))
INSERT INTO big_charges(cust_id,charge_amt)
   VALUES(newrow.cust_id, newrow.charge_amt)
```

Here the incoming transaction's charge amount is checked to see whether it exceeds twice the standard deviation value above the average for this customer, and writes this transaction to the BIG_CHARGES table if true.

Another trigger (not shown here) needs to be written for the BIG_CHARGES table which sets off alarms when the number of such charges over a given period exceeds pre-defined thresholds.

4.3.6 Plot monthly stock prices movement with percentage change

The objective is to determine the average price of the stock per month and its percentage change over the previous month. This information can be used for trend analysis for making investment decisions.

In our example, we use the stock price data for a 6-month period.

Data

The main attributes are stock symbol, date, and closing price of the stock on that date.

BI functions showcased

OVER, ORDER BY, ROWS BETWEEN

Steps

The necessary steps are shown in Example 4-31.

Example 4-31 Monthly movement of stock prices with percentage change

```
WITH month_avg(month,avg_price) AS
   (
   SELECT MONTH(date), AVG(close_price)
   FROM stocktab
   WHERE symbol = 'HAL'
   GROUP BY MONTH(date)
   ),
month_avgs(month,cur_avg,last_avg) AS
   (
   SELECT month,avg_price,MAX(avg_price) OVER (ORDER BY month ROWS BETWEEN
   1 PRECEDING AND 1 PRECEDING)
   FROM month_avg
   )
SELECT month, DEC(cur_avg,7,3) AS "average price",
DEC(100e0*(cur_avg - last_avg)/last_avg,7,3) AS "% Change"
FROM month_avgs
```

The first temporary table created is a straight summation of the averages by month. The second temporary table uses the first to create a table with the additional column showing the preceding month average. Then we select from the second temporary table and calculate the monthly percentage change.

Attention: The ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING clause gets the value of the expression for the immediately preceding row. MAX does not have any impact on a single row.

The results of this query are shown in Figure 4-36.

MONTH	average pri...	% Change
7	11.282	
8	14.673	30.053
9	16.718	13.935
10	11.232	-32.814
11	14.482	28.936
12	16.585	14.517

Figure 4-36 Monthly averages and percent change

This data was copied into Lotus 1-2-3 to produce the bar/line chart shown in Figure 4-37. It shows the average monthly price as line graph and the percentage change as histogram to emphasize the difference between the two types of figures.

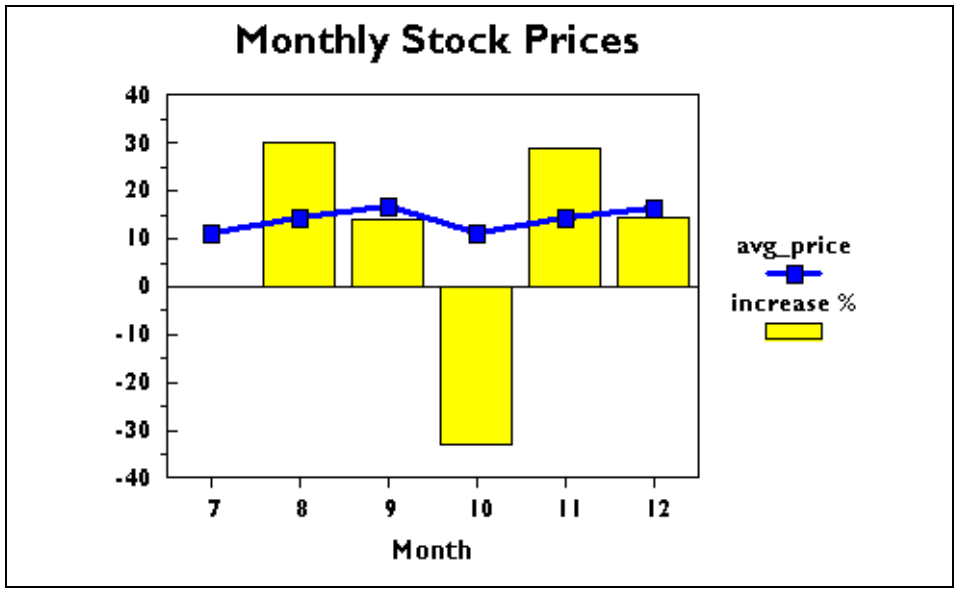


Figure 4-37 Monthly stock prices

4.3.7 Plot the average weekly stock price in September

This time we plot the average weekly price in the month of September, using “running mean” smoothing to reveal underlying trends in the data. The smoothed trend line is sometimes called non-parametric regression because it does not have a parametric representation (e.g., y=ax).

In this example we will use the stock price data for a 3-month period and calculate the moving 7-point average and a calendar week average.

Data

The main attributes are stock symbol, date, and closing stock price on that date.

BI functions showcased

AVG, OVER, ORDER BY, ROWS BETWEEN, RANGE BETWEEN

Steps

The necessary steps are shown in Example 4-32.

Example 4-32 Average weekly stock price in September

```
SELECT date,symbol,close_price,
   DEC(AVG(close_price) OVER (ORDER BY date ROWS BETWEEN 3 PRECEDING and 3
   FOLLOWING),7,3) AS cal_wk_avg,
   DEC(AVG(close_price) OVER (ORDER BY date RANGE BETWEEN 00000003.
   PRECEDING AND 00000003. FOLLOWING)7,3) AS tra_wk_avg
FROM stocktab
```

Note: The result of a DATE arithmetic operation is a DEC(8,0) value. We therefore need to specify the comparison value in the RANGE operator with a precision of DEC(8,0), in order to obtain the correct result.

The first calculation is a straight average of 7 prices - that day's closing price, the 3 preceding recorded prices, and 3 following recorded prices. However, this does not take into account days on which the markets are closed such as holidays and weekends for which there are no prices recorded. ROW based windows are okay when the data is dense; however, when there are missing rows or duplicates, the results can be misleading.

The second calculation uses the range offset to overcome the ROW based window problem. RANGE enables you to specify the aggregation group based in terms of values (date in our case) rather than on absolute row position. Therefore, the running average will only be over the 5 working days, since RANGE over date will limit the aggregation to a maximum of 3 calendar days on either side of the current row's date, and therefore account for the missing weekend prices.

The results of this query are shown in Figure 4-38.

DATE	SYMBOL	CLOSE_PRICE	CAL_WK_AVG	TRA_WK_AVG
2001-09-03	HAL	14.500	17.857	17.550
2001-09-04	HAL	17.000	17.642	17.200
2001-09-05	HAL	17.625	17.178	17.200
2001-09-06	HAL	19.750	16.714	17.200
2001-09-07	HAL	17.125	16.696	17.375
2001-09-10	HAL	15.375	16.339	15.400
2001-09-11	HAL	15.625	16.000	15.025
2001-09-12	HAL	14.375	15.089	15.025
2001-09-13	HAL	14.500	14.696	15.025
2001-09-14	HAL	15.250	14.553	14.625
2001-09-17	HAL	13.375	14.607	14.675
2001-09-18	HAL	14.375	15.000	15.050
2001-09-19	HAL	14.375	15.767	15.050
2001-09-20	HAL	16.000	16.375	15.050
2001-09-21	HAL	17.125	17.232	16.350
2001-09-24	HAL	19.875	17.982	19.100
2001-09-25	HAL	19.500	18.732	19.600
2001-09-26	HAL	19.375	19.187	19.600
2001-09-27	HAL	19.625	19.600	19.600
2001-09-28	HAL	19.625	19.531	19.531

Figure 4-38 September stock prices

The data is copied into Lotus 1-2-3 and charted as shown in Figure 4-39.

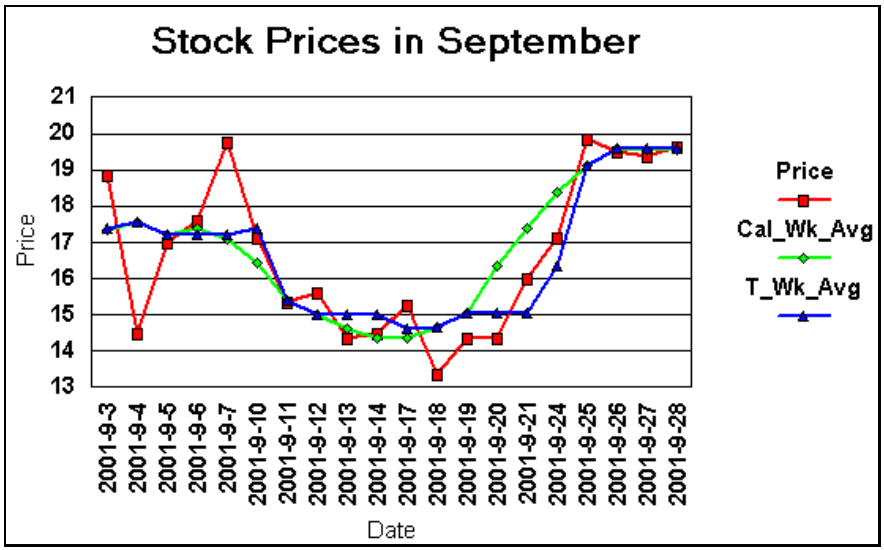


Figure 4-39 September stock prices

4.3.8 Project growth rates of Web hits for capacity planning purposes

An online retailer of books and music recently created a new Web site devoted to discount home electronics, and wants to assess the rate of increase of the Web site hit rate during the first few months of operation. Typically, the daily hit rate for a popular new news site grows rapidly for a while, and so it is natural to try and fit a power (non-linear) curve to the data.

The objective is to project the growth rate in order to perform capacity planning.

Data

The major attributes in this scenario are the number of hits, and the day of the hits on the Web site.

BI functions showcased

REGR_COUNT, REGR_SLOPE, REGR_ICPT

Steps

Consider a non-linear equation of the form:

$$y = bx^a$$

This is equivalent to:

```
log y = a log x + log b
```

It can be represented as shown in Example 4-33.

Example 4-33 Representing a non-linear equation

```
SELECT
   REGR_SLOPE(LOG(y), LOG(x)) AS a,
   EXP(REGR_ICPT(LOG(y), LOG(x))) AS b
   .......
```

The aforementioned curve fitting is explored in the query shown in Example 4-34.

Example 4-34 Computer slope and intercept

```
SELECT
   REGR_COUNT(hits,days) AS num_days,
   REGR_SLOPE(LOG(hits),LOG(days)) AS a,
   EXP(REGR_ICPT(LOG(hits),LOG(days))) AS b
FROM traffic_data
```

The foregoing query lists the number of non-null pairs of hits and day in the table, and computes the values of ‘a’ and ‘b’ in the foregoing equation.

The results of the foregoing query is shown as follows, and indicates that there were 100 non-null data points in the table, and that the estimated values for ‘a’ and ‘b’ are 1.9874 and 21.4302 respectively.

```
            num_days                     a                      b
------------------- ---------------------- ----------------------
                 100                 1.9874                 21.4302
```

R^2 provides the quality of the curve fitting, and is **incorrectly**[2] computed using the SQL shown in Example 4-33.

[2] This is the incorrect method because we are computing this function on the transformed data using logarithmic function, rather than on the untransformed date.

Example 4-35 Compute R^2

```
SELECT
   REGR_COUNT(hits,days) AS num_days,
   DECIMAL(REGR_SLOPE(log(hits), log(days)),10,4) AS a,
   DECIMAL(EXP(REGR_ICPT(log(hits), log(days))),10,4) AS b,
   DECIMAL(REGR_R2(log(hits), log(days)),10,4) AS r2
FROM traffic_data
```

The results of the foregoing query, using the built-in R2 function, are shown as follows.

```
      num_days             a               b               r2
-------------- ------------- --------------- ----------------
           100        1.9874         21.4302           0.9912
```

However, in order to correctly compute R^2 for the non-linear fit of the original untransformed data, the SQL shown in Example 4-36 should be used.

Example 4-36 Correct R^2 *computation on original untransformed data*

```
WITH coeffs(a,b) AS
   (
   SELECT
      REGR_SLOPE(LOG(hits),LOG(days)) AS a,
      EXP(REGR_ICPT(LOG(hits),LOG(days))) AS b
   FROM traffic_data
   ),
residuals(days,hits,error) AS
   (
   SELECT
      t.days,t.hits,t.hits - c.b*power(t.days,c.a)
   FROM traffic_data t, coeffs c
   )
SELECT 1e0-(SUM(error*error)/REGR_SYY(hits,days)) AS rr2
FROM residuals
```

The result of this query is as follows:

```
-------------------rr2
+9.55408646608249E-001
```

Note that the correct value R^2 is 0.955 which is lower than 0.991. This is typical. Computing of R^2 for the transformed data usually results in an overestimate of the goodness of fit.

The curve fitting data and R^2 value is shown in Figure 4-40 as charted by Kaleidograph.

Caveat: Such a curve is useful for short range predictions, but not long range ones.

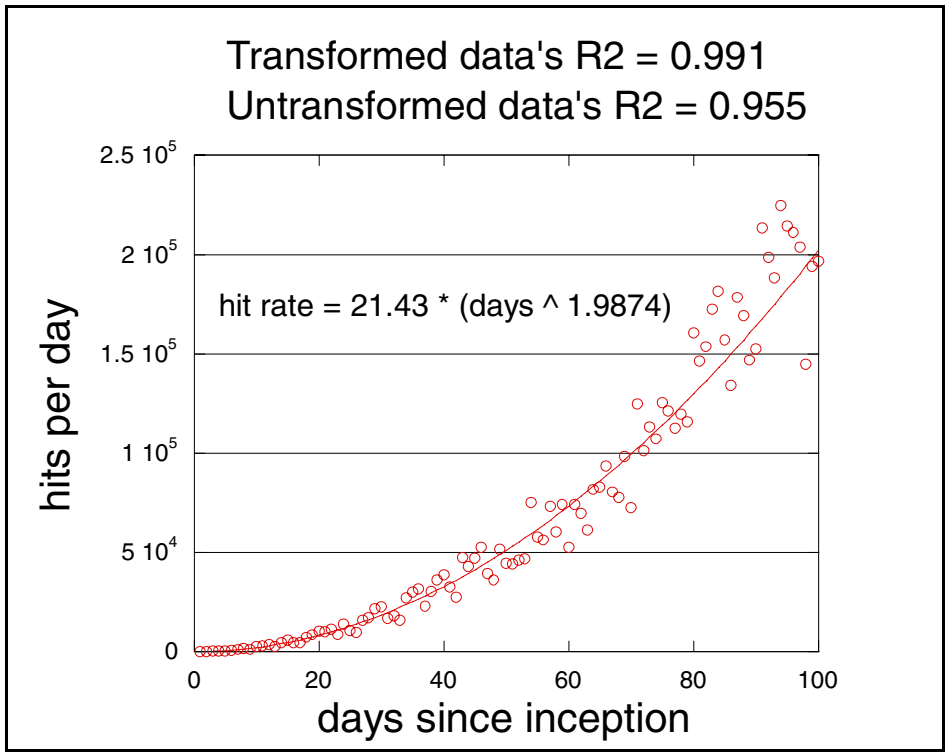


Figure 4-40 Non-linear curve fitting

The graph demonstrates that the hit rate grows non-linearly as per the following equation:

Hit Rate = 21.43 * $days^{1.9874}$

4.3.9 Relate sales revenues to advertising budget expenditures

The objective is to identify the impact of advertising budget expenditures on sales revenues in a number of cities in California, and potentially interpolate the perceived relationship via a regression.

We answer this question in 2 phases as follows:

- In phase 1, we use the HAT diagonal to determine the quality of the sample data before proceeding with the regression analysis.
- In phase 2, we perform regression on the sound sample data and use standard deviation about the regression line to identify unusually effective advertising campaigns.

Phase 1

In order to perform a good regression analysis, it is important that the sample data should be spread out evenly over the range of the dependent variable, otherwise, some data may have an undue influence over the regression line parameters. This is investigated via the HAT diagonal for the sample cities.

Data

The main source of data for this query is a table containing the advertising expenditures and sales revenues for each city.

BI functions showcased

REGR_AVGX, REGR_SXX

Steps

We executed the SQL shown in Example 4-37 to determine the HAT Diagonal for the set of various cities:

Example 4-37 Determine the HAT Diagonal for the set of various cities

```
WITH stats(mx, mx2, sxx) AS
   (
   SELECT REGR_AVGX(sales,ad_budget),REGR_AVGX(sales,ad_budget*ad_budget),
      REGR_SXX(sales,ad_budget)
   FROM cal_ad_camp
   )
SELECT d.label as city,
   (s.mx2 - 2 * s.mx * d.ad_budget + d.ad_budget * d.ad_budget) / s.sxx
      AS hat
FROM cal_ad_camp d, stats s
ORDER BY hat DESC
```

Note that we use REGR_AVGX(sales,ad_budget*ad_budget) rather than AVG(ad_budget*ad_budget) for two reasons:

1. AVG(ad_budget*ad_budget) may include x-values for which the corresponding y-value is NULL, which is not used in the regression.
2. Regression functions are designed to work together, efficiently computing all the necessary statistics in a single pass through the data.

The results of the foregoing query are shown below and charted in Figure 4-41.

```
city     hat
-----------  ------
Los Angeles  0.9644
Boonville    0.1222
Grass Valley 0.1195
Yreka        0.1154
Gilroy       0.1099
Lemoore      0.1011
Hilmar       0.1011
Mendocino    0.0923
Turlock      0.0922
Morgan Hill  0.0910
Truckee      0.0910
```

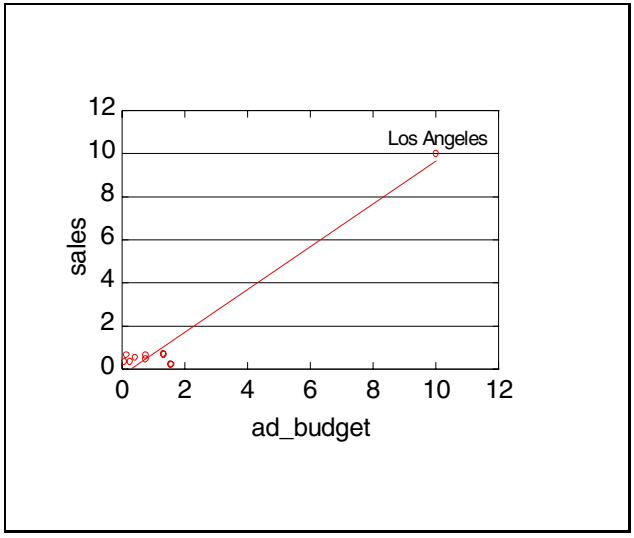


Figure 4-41 Hat diagonal

The results show that right most x_axis advertising budget expenditures corresponding to Los Angeles has a significant impact on the slope of a potential regression line.

This draws attention to the fact that this outlier merits further investigation. In this case our domain knowledge of Los Angeles and the other cities such as Morgan Hill, tells us it is the relative population difference and number of stores that accounts for this outlier. In other cases, it may point to a data capture error. The HAT Diagonal thus highlights outliers that merit further investigation by domain experts.

Attention: We therefore strongly recommend computing the HAT diagonal on the available data points **before** attempting a regression on it. The existence of outliers influencing the slope may draw attention to the need for further investigation about the quality of the data.

Phase 2

We determine the effectiveness of the advertising budget on sales in this phase using a demographically similar sample.

Data

The source of data is the cleaned up sample of Phase 1.

BI functions showcased

REGR_SLOPE, REGR_ICPT, REGR_COUNT, REGR_SXX, REGR_SYY, REGR_SXY

Steps

We executed the SQL shown in Example 4-38 to identify cities where the deviation of advertising budgets to sales revenues were outside the norm by more than two standard deviations above or below the regression line, where “standard deviation” is defined in a manner appropriate for regression. The regression line represents the “normal” relation between advertising budgets and resulting sales revenues.

Example 4-38 Cities where budgets to sales deviations outside the norm

```
WITH dt(a, b, sigma) AS
   (
   SELECT
      REGR_SLOPE(sales,ad_budget),
      REGR_ICPT(sales,ad_budget),
      SQRT((REGR_SYY(sales,ad_budget)-(REGR_SXY(sales,ad_budget)
       *REGR_SXY(sales,ad_budget)
       /REGR_SXX(sales,ad_budget)))
       / (REGR_COUNT(sales,ad_budget) - 2))
   FROM ad_camp
   )
SELECT city, ad_budget, sales
```

```
FROM ad_camp ac, dt
WHERE sales > a*ad_budget + b + 2e0*sigma
ORDER BY ad_budget
```

The foregoing query computes the “regression standard deviation” **sigma** in terms of build-in regression functions. The results of the foregoing query are shown below and charted in Figure 4-42.

```
CITY       BUDGET   SALES
------     ------   ------
Fresno      15.26    82.00
San Diego   84.99   223.81
```

Here, San Diego and Fresno are cities that show their sales revenues being two or more standard deviations about the norm as represented by the regression line. Fresno obviously showed significantly better sales revenue returns on adverting budget investment as compared to the norm, while San Diego did not.

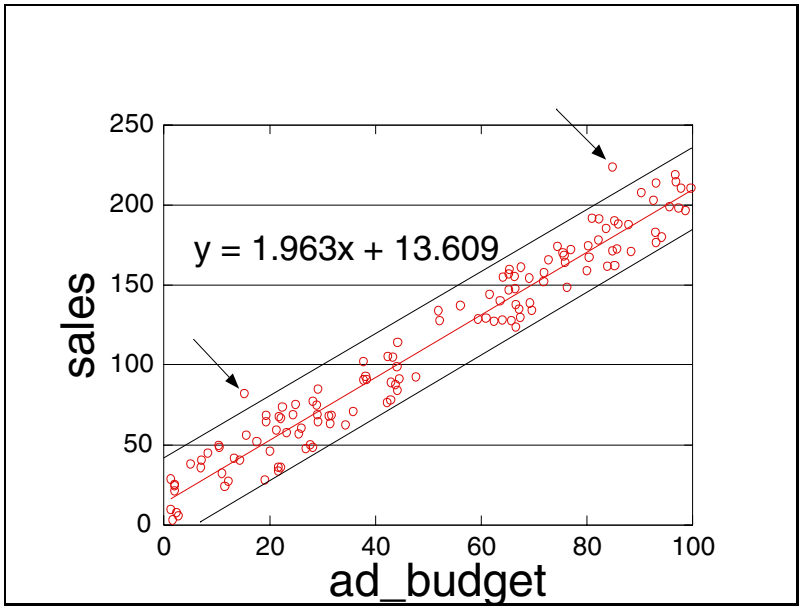


Figure 4-42 Standard deviation around regression line

4.4 Sports

We have selected the following typical sports queries for our examples.

1. For a given sporting event:
   a. List all the athletes competing in the event
   b. Rank the athletes by score and identify the medal winners
   c. Rank each athlete within their individual countries
   d. Identify the medals won by country, and the total number of medals awarded to date
   e. List the medals won by day, country and the total number of medals won by each country
   f. Rank the athletes when there are tied scores
2. Seed the players at Wimbledon.

Acknowledgement: Much of the data and the SQL examples shown here have been adapted from Dan Gibson of IBM Toronto Lab's paper on "DB2 Universal Database's Business Intelligence Functions assist in the Sydney 2000 Olympics Games".

4.4.1 For a given sporting event

We review the diving event in our example.

Data

The key attributes of interest are:

- Event identification
- Name of athlete
- Country athlete represents
- Score of athlete in diving

BI functions showcased

RANK, DENSERANK, OVER, PARTITION BY, ORDER BY, ROLLUP, CUBE

Steps

Query a

List all the athletes competing in the event, as shown in Example 4-39.

Example 4-39 All athletes competing in the event

```
SELECT event_name, event_date, athlete, country, score
FROM event
```

The results of this query are shown in Figure 4-43.

EVENT_NAME	EVENT_DATE	ATHLETE	COUNTRY	SCORE
Diving	2000-01-15	Dan	Canada	10.0
Diving	2000-01-15	Dave	USA	9.9
Diving	2000-01-15	Rob	USA	9.8
Diving	2000-01-15	Grant	Australia	9.7
Diving	2000-01-15	Gene	UK	9.9
Diving	2000-01-15	Leon	UK	7.0
Diving	2000-01-15	Ed	Brazil	9.0
Diving	2000-01-15	Bob	Brazil	9.5
Diving	2000-01-15	Bob	Brazil	9.5
Gymnastics	2000-01-16	Jack	Brazil	8.0
Gymnastics	2000-01-16	Chris	Australia	9.8
Gymnastics	2000-01-16	Tim	South Africa	9.0
Gymnastics	2000-01-16	Bill	Romania	10.0

Figure 4-43 All athletes in the diving event

Query b

Rank the athletes by score and identify the medal winners, as shown in Example 4-40.

Example 4-40 Rank athletes by score and identify the medal winners

```
SELECT event_name, athlete, score, country,
   DENSERANK() OVER(ORDER BY score DESC) AS Rank,
   CASE DENSERANK() OVER(ORDER BY score DESC)
      WHEN 1 THEN 'Gold'
      WHEN 2 THEN 'Silver'
      WHEN 3 THEN 'Bronze'
      END AS Medal
FROM event
WHERE event_name='Diving'
ORDER BY Rank
```

The results of this query are shown in Figure 4-44.

EVENT_NAME	ATHLETE	SCORE	COUNTRY	RANK	MEDAL
Diving	Dan	10.0	Canada	1	Gold
Diving	Dave	9.9	USA	2	Silver
Diving	Gene	9.9	UK	2	Silver
Diving	Rob	9.8	USA	3	Bronze
Diving	Grant	9.7	Australia	4	
Diving	Bob	9.5	Brazil	5	
Diving	Bob	9.5	Brazil	5	
Diving	Ed	9.0	Brazil	6	
Diving	Leon	7.0	UK	7	

Figure 4-44 Gold, Silver and Bronze winners in diving

The query was run in IBM QMF and results saved in an MS-Excel spreadsheet, and then bar-charted as shown in Figure 4-45.

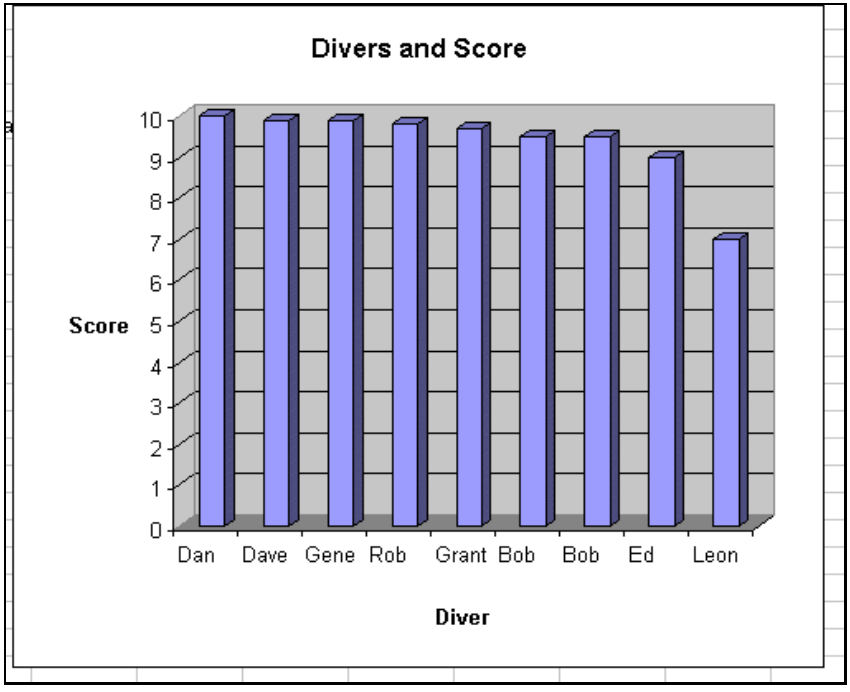


Figure 4-45 Divers and their scores

Query c

Rank each athlete within their individual countries, as shown in Example 4-41.

Example 4-41 Rank each athlete within their individual countries

```
SELECT athlete, score, country, rank FROM
(
SELECT event_name, athlete, score, country,
DENSERANK () OVER(PARTITION BY country ORDER BY score DESC) as Rank
FROM event
) AS nested_events
```

The results of this query are shown in Figure 4-46.

ATHLETE	SCORE	COUNTRY	RANK
Chris	9.8	Australia	1
Grant	9.7	Australia	2
Bob	9.5	Brazil	1
Bob	9.5	Brazil	1
Ed	9.0	Brazil	2
Jack	8.0	Brazil	3
Dan	10.0	Canada	1
Bill	10.0	Romania	1
Tim	9.0	South Africa	1
Gene	9.9	UK	1
Leon	7.0	UK	2
Dave	9.9	USA	1
Rob	9.8	USA	2

Figure 4-46 Athletes ranking in their country

Query d

Identify the medals won by country, and the total number of medals awarded to date, as shown in Example 4-43.

Example 4-42 Medals by country & total medals awarded to date

```
WITH medal_info (day, score, country, event, medal) AS
   (
   SELECT DAYNAME(event_date), score, country, event_name,
      CASE DENSERANK() OVER(PARTITION BY event_name ORDER BY score DESC)
      WHEN 1 THEN 'Gold'
      WHEN 2 THEN 'Silver'
      WHEN 3 THEN 'Bronze'
```

```
      END AS medal
   FROM event
   )
SELECT day, country, COUNT(medal) AS Medal
FROM medal_info
WHERE medal IS NOT NULL
GROUP BY ROLLUP(day, country)
ORDER BY day, country
```

The predicate “WHERE medal IS NOT NULL” ignores athletes who did not get medals.

The results of this query are shown in Figure 4-47.

DAY	COUNTRY	MEDAL
Saturday	Canada	1
Saturday	UK	1
Saturday	USA	2
Saturday		4
Sunday	Australia	1
Sunday	Romania	1
Sunday	South Africa	1
Sunday		3
		7

Figure 4-47 Number of medals each country won and total medals awarded v.1

A slight variation of the foregoing query has the order changed from “day, country” to “country, day”. Notice that, in Example 4-43, totals by country are returned rather than totals by day!

Example 4-43 Medals by country by day

```
WITH medal_info (day, score, country, event_name, medal)AS
   (
   SELECT DAYNAME(event_date), score, country, event_name,
      CASE DENSERANK() OVER(PARTITION BY event_name ORDER BY SCORE DESC)
      WHEN 1 THEN 'Gold'
      WHEN 2 THEN 'Silver'
      WHEN 3 THEN 'Bronze'
      END AS medal
   FROM event
   )
SELECT day, country, COUNT(medal) AS Count
```

```
FROM medal_info
WHERE medal IS NOT NULL
GROUP BY ROLLUP(country, day)
ORDER BY country, day
```

The results of this query are shown in Figure 4-48.

DAY	COUNTRY	COUNT
Sunday	Australia	1
	Australia	1
Saturday	Canada	1
	Canada	1
Sunday	Romania	1
	Romania	1
Sunday	South Africa	1
	South Africa	1
Saturday	UK	1
	UK	1
Saturday	USA	2
	USA	2
		7

Figure 4-48 Number of medals each country won and total medals awarded v.2

Query e

List the medals won by day, country and the total number of medals won by each country, as shown in Example 4-44.

Example 4-44 Medals won by day, country and total medals by country

```
WITH medal_info (day, score, country, event, medal) AS
   (
   SELECT DAYNAME(event_date), score, country, event_name,
      CASE DENSERANK() OVER(PARTITION BY event_name ORDER BY score DESC)
      WHEN 1 THEN 'Gold'
      WHEN 2 THEN 'Silver'
      WHEN 3 THEN 'Bronze'
      END AS medal
   FROM event
```

```
   )
SELECT day, country, count(medal) AS Count
FROM medal_info
WHERE medal IS NOT NULL
GROUP BY CUBE(day, (country,medal))
ORDER BY day, country
```

The results of this query are shown in Figure 4-49.

DAY	COUNTRY	COUNT
Saturday	Canada	1
Saturday	UK	1
Saturday	USA	1
Saturday	USA	1
Saturday		4
Sunday	Australia	1
Sunday	Romania	1
Sunday	South Africa	1
Sunday		3
	Australia	1
	Canada	1
	Romania	1
	South Africa	1
	UK	1
	USA	1
	USA	1
		7

Figure 4-49 Medals won by day, country and total medals by country

Query f

Rank the athletes when there are tied scores, as shown in Example 4-45.

Example 4-45 Rank athletes when scores are tied

```
SELECT event_name, athlete, score, country,
   RANK() OVER(ORDER BY score DESC) AS Gappy_Rank
FROM event
WHERE event_name='Diving'
ORDER BY Gappy_Rank
```

The results of this query are shown in Figure 4-50.

EVENT_NAME	ATHLETE	SCORE	COUNTRY	GAPPY_RANK
Diving	Dan	10.0	Canada	1
Diving	Dave	9.9	USA	2
Diving	Gene	9.9	UK	2
Diving	Rob	9.8	USA	4
Diving	Grant	9.7	Australia	5
Diving	Bob	9.5	Brazil	6
Diving	Bob	9.5	Brazil	6
Diving	Ed	9.0	Brazil	8
Diving	Leon	7.0	UK	9

Figure 4-50 Ranking when there are ties

4.4.2 Seed the players at Wimbledon

Many tournaments organize their knockout competitions based on seedings. Seeding of players is designed to prevent the leading players from meeting one another in the early rounds of a tournament.

Previously at tournaments like Wimbledon the seeding had been decided by a committee. Nowadays, they are based on calculations based on the players' recent performances at pre-defined tournaments.

While a professional ranking and the seeding for a tournament are interrelated, they are not the same thing. A professional ranking is normally an indication of performance over a set period of time and a specific set of competitive events. Seeding is usually done using the ranking and information specific to the event, and is meant to insure the best draw possible.

In preparing a draw, it is important to consider as much background information about the player/team's performance as possible. Most events are seeded according to a set of criteria that current ranking, previous performance in the competition or similar competitions.

Data

The major attributes in this scenario are current ranking, and the participants results from the last 5 tournaments.

BI functions showcased

RANK, DENSERANK

Steps

The necessary steps are shown in Example 4-46.

Example 4-46 Seed the players at Wimbledon

```
WITH tt1 (player,rank,r1,r2,r3,r4,r5) AS
   (
   SELECT player,w_ranking,DENSERANK() OVER (ORDER BY t_1 ASC),
      DENSERANK() OVER (ORDER BY t_2 ASC),
      DENSERANK() OVER (ORDER BY t_3 ASC),
      DENSERANK() OVER (ORDER BY t_4 ASC),
      DENSERANK() OVER (ORDER BY t_5 ASC)
   FROM seedings
   ),
   tt2 (player,rank,seeding) AS
      (
      SELECT player,rank,RANK() OVER (ORDER BY rank+r1+r2+r3+r4+r5, rank
      ASC)
      FROM tt1
      )
SELECT player,rank,seeding
FROM tt2
WHERE seeding <= 16
ORDER BY seeding
```

This query creates a temporary table “dt” with results from previous tournaments converted to a numerical value based on the DENSERANK function (this replaces null with the lowest rank). The “dt” table is then used to calculate the seeding for each player. When seeding is duplicate, weighting is applied based on the player’s world ranking and only the top 16 players are seeded.

The results of this query are shown in Figure 4-51.

PLAYER	W_RANKING	SEEDING
Ferrari J	5	1
August A	3	2
Simper P	10	3
Howitt L	1	4
Cockeril T	9	5
Kaffel Y	4	6
House T	8	7
Reddick A	14	8
Curtain G	2	9
Grossman S	6	10
Beam P	7	11
Costa A	16	12
Gambol J	13	13
Arturo W	11	14
Evanic G	19	15
Ljubicic R	20	16

Figure 4-51 Tournament seeding

This data can be copied into Lotus 1-2-3 and charted as show in Figure 4-52.

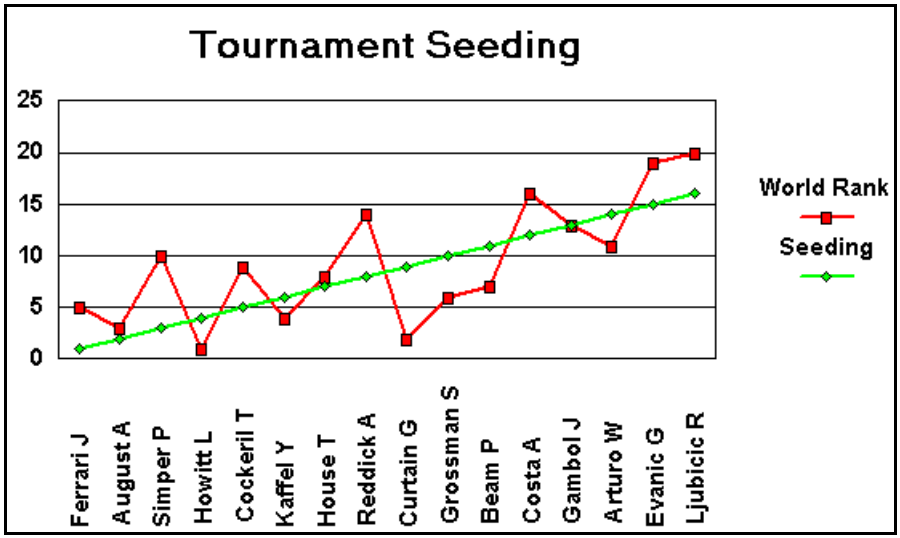


Figure 4-52 Comparison graph to demonstrate seeding versus world rank

The graph demonstrates that the weighting added by the individual's performance in previous tournaments would produce a different seeding to the world ranking. In fact in this scenario only one player (T House) has a seeding exactly the same as his ranking.

A

Introduction to statistics and analytic concepts

In this appendix we offer an introduction to various statistics and analytic concepts used in the earlier chapters to solve business problems.

The audience for this section consists of application developers, as well as DBAs, who are required to translate the power users' business problems into one or more SQL queries that return the required result using the built-in analytic functionality of DB2 UDBs.

A.1 Statistics and analytic concepts

The following statistics and analytic concepts are introduced in this section:

- Variance
- Standard deviation
- Covariance
- Correlation
- Regression
- Hypothesis testing
- Hat diagonal
- Wilcoxon rank sum test
- Chi-Squared test

In the next several sections we will briefly discuss these terms.

A.1.1 Variance

Variance measures the spread of a set of observations/measurements around the average value for this set. In other words, it is the average squared deviation of a set.

For example, assume 10 employees in department D1 having salaries as shown in Table A-1.

Table A-1 Salaries for department D11

Name	Salary (in $000)
Mary	40
Bill	35
Dan	50
Jean	45
Gene	43
Sally	60
Michael	30
Robert	44
Samantha	48
John	45

In order to determine the variance in these salaries, we first need to determine the average salary. Average is also referred to as the mean, and the terms are interchangeable for our purposes.

The average is ((40,000 +35,000+...+48,000+45,000)/10) = 44,000.

Variance is defined as:

$$Var(x) = \left(\sum_{i=1}^{n} (X_i - \bar{X})^2 \right) / n$$

Where:

X_i is the i*th* observation on variable X.

X is the average.

i starts at 1 and continues up to n observations.

The Greek letter Sigma represents the summary of the enclosed equation.

Important: When the data values represent the entire set of values, then the *Population Variance* is computed.

When the data values represent a sample (subset) of the entire set of values, then the *Sample Variance* is computed.

The relationship between population variance (Var_{pop}) and sample variance (Var_{samp}) is as follows:

$$Var_{pop} = \frac{(n-1)}{n} \times Var_{samp}$$

Where:

n is the population size

A.1.2 Standard deviation

While the computation of variance is necessary and useful for further analysis of data, its meaning is most often not intuitive. The better understood quantity is standard deviation.

Standard deviation is the root mean square distance of the data from the mean. The definition of standard deviation is the square root of the variance.

$$StdDev = SQRT(VAR(X)) = \sqrt{\left(\sum_{i=1}^{n} (X_i - \overline{X})^2\right) / n}$$

Important: When the data values represent the entire set of values, then the *Population Standard Deviation* is computed.

When the data values represent a sample (subset) of the entire set of values, then the *Sample Standard Deviation* is computed.

The relationship between population standard deviation (SD_{pop}) and sample standard deviation (SD_{samp}) is as follows:

$$SD_{pop} = \sqrt{\frac{(n-1)}{n}} \times SD_{samp}$$

Where:

n is the population size.

A.1.3 Covariance

This is not related to the variance function.

Covariance is a measure of the linear association between two variables. The The covariance value depends upon the units of measurement of the variables involved, and therefore unusable directly. A more useful measure of the linear relationship can be gained via correlation.

Covariance is defined as follows:

$$Cov(X, Y) = \sum_{i=1}^{n} ((X_i - \overline{X}) \times (Y_i - \overline{Y})) / n$$

Where:

X_i is the i*th* observation on variable X.

Y_j is the j*th* observation on variable Y.

$\bar{X}$ is the average of all values of X.

$\bar{Y}$ is the average of all values of Y.

i starts at 1 and continues up to n observations.

The Greek letter Sigma represents the summary of the enclosed equation.

The meaning of covariance given Table A-2.

Table A-2 Covariance meaning

Covariance Value	Meaning
Greater than zero (positive)	The variables are directly linearly related. As one increases so does the other.
Zero	There is no linear relationship between the two variables.
Less than zero (negative)	The variables are inversely linearly related. As one increases the other decreases.

Important: When the data values represent the entire set of values, then the *Population Covariance* is computed.

When the data values represent a sample (subset) of the entire set of values, then the *Sample Covariance* is computed.

The relationship between population covariance ($Covar_{pop}$) and sample covariance ($Covar_{samp}$) is as follows:

$$Covar_{pop} = \frac{(n-1)}{n} \times Covar_{samp}$$

Where:

n is the population size.

A.1.4 Correlation

Correlation is normalized covariance.

That is divide the covariance by the square root of the variance of the two variables as represented by the following equation:

$$CORR(X, Y) = Cov(X, Y) \div (\sqrt{Var(X) \times Var(Y)})$$

The correlation value (most texts refer to it as a coefficient) is a number between -1 and 1.

The values for CORR(X,Y), the correlation coefficient is shown in Table A-3.

Table A-3 Correlation coefficient meaning

Correlation Value	Meaning
CORR(X,Y) between 0 and 1 (positive)	The attributes are directly linearly related. As one increases so does the other.
CORR(X,Y) equal 0	There is no linear relationship between the two attributes.
CORR(X,Y) between -1 and 0 (negative)	The attributes are inversely linearly related. As one increases the other decreases.

Correlation coefficients measure the degree of the linear relationship between variables.

- **+1** indicates a very strong or direct relationship. If we plotted the salary and bonus data that had a correlation of 1 *all* the data points would lie on a straight line with positive slope.
- **-1** also indicates a strong relationship, and the data points would also lie in a straight line but with negative slope.
- **0** indicates that there is no linear relationship and the data points are scattered all over the place.

Therefore, correlation is a measure of how well the data points align themselves. Figure A-1 is a visual representation of varying correlation coefficients.

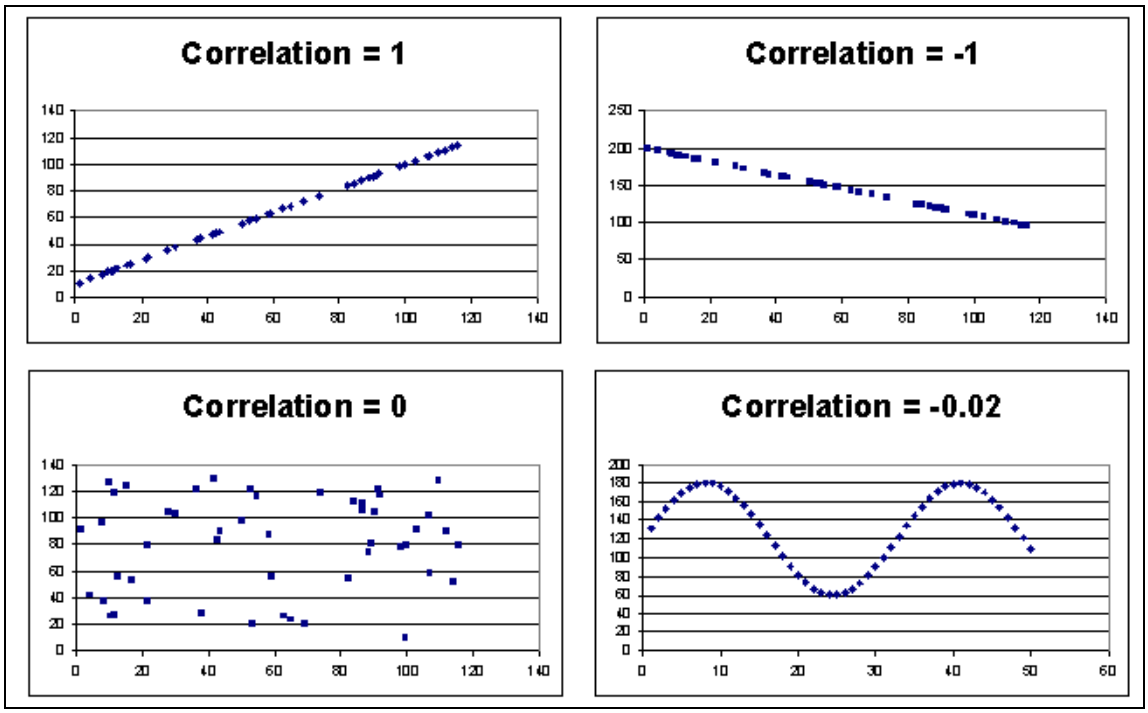


Figure A-1 Sample correlation visualizations

Note: Correlation does *not* identify any non-linear relationships that may exist between two sets of variables. The visualization graph with correlation = -0.02 shows a strong sinusoidal relationship that correlation does not identify.

A.1.5 Regression

The purpose of regression is to find the linear equation that best describes the relationship between a dependent variable, and two or more dependent variables. This equation can then be used to interpolate or extrapolate unobserved/unmeasured values, or detect outliers.

When there is one dependent and one independent variable the regression is called a simple linear regression. Some texts also refer to this method as *Least Squares Fit.*

A linear relationship can be expressed in the form:

$$y = Ax + B$$

Where:

x is an independent variable.

A is the slope of the line.

B is the intercept with the y-axis.

y is the dependent variable.

A simple regression tries to find the best fit for A and B given a number of data points.

Once the model coefficients (A and B) are computed, we need to know the accuracy of the model This is critical in order to understand the reliability of any extrapolations.

The coefficient of determination or r squared ($\mathbf{r}^2$) value provides us with information about the accuracy of the linear regression model that was computed.

A.1.5.1 R^2 (R Squared)

The efficacy of the regression model is often measured by how much of the variability of the data it explains.

R^2 (also known as the coefficient of determination) can be interpreted as the proportion of variability in the data values that is explained by the model.

The value of R^2 ranges between 0 and 1. The closer it is to 1, the better the model does in explaining the data.

A.1.6 Hypothesis testing

Hypothesis testing is a major part of inferential statistics. It is a formal procedure to collect sample data and then use this data to verify whether a given hypothesis is true or not.

Attention: A hypothesis is a claim or statement about the state of the world.

Null hypothesis is the logical negation of the hypothesis.

The hypothesis often takes the form of a statement about an unknown population[1] parameter, or the relation between unknown population parameters.

A hypothesis test begins with two statements about a population that are mutually exclusive:

1. The average weight of mountain lions is 150 pounds.
2. The average weight of mountain lions is not 150 pounds.

Often the statements will refer to a population parameter such as a population mean. Sometimes it applies to more than one population, such as a claim that the means of 4 different populations are all equal.

Since the population parameter is a number (call it PP), these statements will have one of the following three different forms, where ‘a’ is a constant.

1. PP is equal to ‘a’ versus PP is not equal to ‘a’.
2. PP is greater than or equal to ‘a’ versus PP is less than ‘a’.
3. PP is less than or equal to ‘a’ versus PP is greater than ‘a’.

The hypothesis that includes equality is called a **null** hypothesis, while the one that does not include equality is called the **alternative** hypothesis.

In each of the above three forms listed above, the first statement is a **null** hypothesis.

The sample data provides the way to distinguish between the null and alternative hypothesis. For example, if the null hypothesis claims that the population mean is 10, while the sample mean turns out to be 5 and the sample data is very representative of the population, then the odds are good that the null hypothesis is wrong. Likewise, if the claim is that the population mean is greater than or equal to 10, and the sample mean is 5, then the odds are good that the null hypothesis is wrong, and similarly for the third form listed above.

Important: In the case of an equality hypothesis, the sample data will rarely prove the null hypothesis to be true. It will be very difficult to convince someone that the population mean is exactly 10 no matter how much of a sample we gather.

If we do not reject the null hypothesis, we accept it rather than affirm it.

[1] A population is a collection of all data points of interest.

In order to measure whether the sample data is extreme enough to contradict the null hypothesis, a test statistic is used. This is a random variable with a known probability distribution that can be used to measure how likely we are to get such sample data given that the null hypothesis is true. If, given the null hypothesis, the probability of getting such a value is extremely low, then we would be inclined to reject the null hypothesis in favor of the alternative hypothesis. On the other hand, if the probability is not too small, then the null hypothesis might well be true and we would have to accept it.

Some well known test statistics include the chi-squared statistic, and the Wilcoxon Rank Sum Test 'W' statistic.

Before computing the test statistic, a **significance level** needs to be set. This is our cutoff in terms of what we consider to be a probability that could happen by chance, and typically is either 5% or 1%. The probability that given the null hypothesis, you would get sample data this extreme or worse is called the **p-value** of the test. Once the **p-value** is found, it is compared to the **significance level**. If the **p-value** is larger than the **significance level**, then you accept the null hypothesis. If the **p-value** is less than the **significance level**, then you reject the null hypothesis.

In the first case (PP is equal to 'a'), the null hypothesis can be proved wrong in two ways as follows:

1. PP is bigger than 'a'.
2. PP is smaller than 'a'.

This kind of test is called a **two-tailed hypothesis test**, because in the graph of the probability distribution of the test statistic, the "tails" to both the left and right correspond to the rejection of the null hypothesis.

In the second case, where rejection occurs because we think that PP is smaller than the constant 'a' is called a l**eft-tailed hypothesis test**.

In the third case, where rejection occurs because we think PP is larger than the constant 'a' is called a **right-tailed hypothesis tes**t.

Collectively the left-tailed and right-tailed hypotheses tests are called **one-tailed tests**.

A.1.7 HAT diagonal

The HAT diagonal is used in conjunction with linear regression.

As discussed, linear regression involves a best fit of a collection of x,y pairs to a mathematical equation of the form:

$$y = Ax + B$$

However, depending upon the data points, it is possible for the slope A and intercept B to be unduly influenced by data points far from the mean of x.

The HAT diagonal test measures the leverage of each observation on the predicted value for that observation.

This concept is demonstrated in Figure A-2, “HAT diagonal influence of individual data points” on page 227. The data point at x=9 is far from the data points with X-values between 0 and 2. The paired value of y for x=9 has a large influence on the slope in this example.

Therefore, depending upon the computed slope, for a given value of x=9, the value of Y is 5 or 9 which is a significant difference.

The HAT diagonal test identifies data paris that exert such undue influence on the computed slope. The user may then choose to exclude or include these data points from the computed linear regression model based on their unique understanding of the domain of these data points, in order to obtain a more accurate linear regression model.

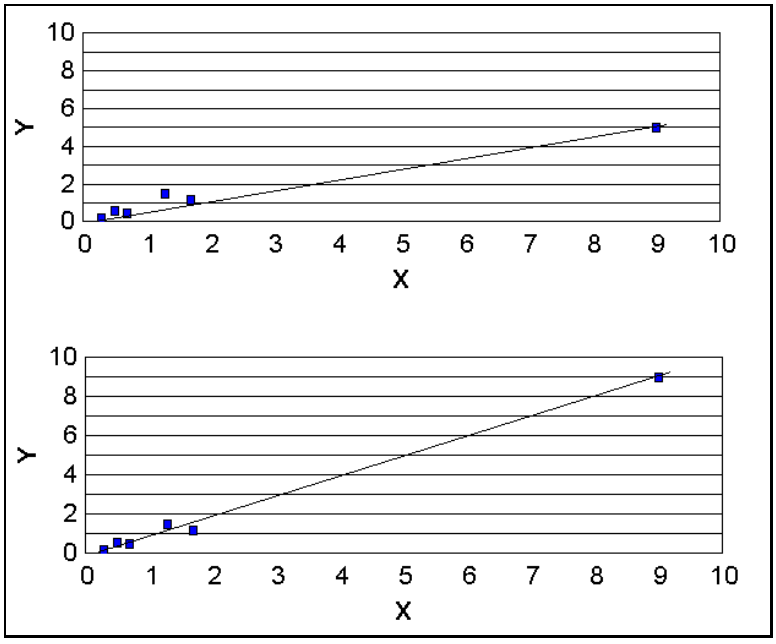


Figure A-2 HAT diagonal influence of individual data points

The formula to calculate the HAT diagonal is:

$$HATdiagonal(H_i) = (m_{x^2} - (2m_x x_i + x^2{}_i))/(S_{xx})$$

Where:

$$m_{x^2} = \frac{1}{n}\sum_{i=1}^{n} x_i^2$$

$$m_x = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$S_{xx} = \sum_{i=1}^{n} (x_i - m_x)^2$$

Values that are above the generally accepted cutoff of **2*p*/*n*** for the HAT diagonal values, should be investigated further to determine their validity. This is typically done by including more values in the regression to represent this outlying range or validation of this data pair. Here:

- ***n*** is the number of observations used to fit the model.
- ***p*** is the number of parameters in the model.

For the charts in Figure A-2 on page 227:

- p is 1 variable.
- n is 6 observations.

Therefore, any observation whose HAT is greater than 2x1/6 = 0.33, should be suspect.

The data point at x=9 has a HAT of 0.979 which is nearly three times larger than the cutoff, and is therefore suspect.

A.1.8 Wilcoxon rank sum test

The Wilcoxon rank sum test is used to test the assumption that outcomes for a variable (or a set of variables) under differing conditions are related. That is, the outcome under the test and control conditions are purely random and the two conditions have no effect on the outcome.

The basic concept here is that if there is no difference between attribute values for two populations, then ranks from one population should not be systematically higher or lower than those of the other. The distribution of ranks for a population is known under null hypothesis. If 'W' is so large that the probability of 'W' being greater than or equal to the computed statistic is small, then reject the null hypothesis.

A potential use of this test is determining the success or failure of a marketing campaign in a particular demographic area by comparing the sales results of a campaign test area with that of another area where no campaign was in effect.

The test methodology involves:

1. Determining an overall ranking to all the outcomes in both test and control groups
2. Summing the ranks of the test group and comparing it with an expected result sum of ranks assuming there is no effect in the test group. The expected results is based on the number of observations made in both the test and control group.
3. Looking up in published tables the probability that the expected and observed results differ by some value.

Important: The Wilcoxon test is attractive because it is a non-parametric test. In other words, it does not require that the distribution of the attribute values have a specified functional form such as a normal distribution, gamma distribution, etc. Other tests like the TWO SAMPLE t TEST are parametric tests which make assumptions that the data is normally distributed. The absence of these limitations makes the Wilcoxon test attractive.

A.1.9 Chi-Squared test

The Chi-squared ($\mathbf{X^2}$) test is used to determine if two or more "categorical" attributes are independent.

A categorical attribute has a finite number of possible values. An example of a categorical attribute could be gender. There are only two possible values, male or female. Another example is the answer to a survey question where the only allowed responses are satisfied, neutral, or dissatisfied.

The basic concept here is that if two attributes are independent, then the expected frequencies factor is defined by the probability equation as follows:

```
P(a,b) = P(a) * P(b)
```

where 'P' is the probability, and 'a" and 'b' represent the attributes.

We then see if the actual frequencies approximately satisfy the above rule.

The formula for $\mathbf{X^2}$ is:

$$ChiSquared(X^2) = \sum (O - E)^2 / E$$

Where:

O is the observed frequency

E is the expected frequency.

Clearly, to use $\mathbf{X^2}$ one must determine or be given the expected frequency of a given attribute having a specific outcome.

- In a coin flip example, the expected frequency of a head or tail being flipped is equal given there are no external forces in play on our coin. Thus the expected frequency is 50% for heads and 50% for tails.
- Another example of expected frequency where there are only two possible outcomes but the expected frequencies of either outcome is not equal is human male/female population categorization for different age ranges. Younger populations are nearly equal in frequency of males and females, while older populations tend to have high frequencies of females than males. Clearly this is due to females living longer than males.

Once we have the expected frequency and have our observed data categorized, $\mathbf{X^2}$ requires the calculation of our observed frequency for each category.

a. After that the difference between observed and expected frequencies is squared so that all values are positive.

a. Sum these values and calculate the ratio of this sum against the expected frequency.

The closer this value is to zero the more likely the attributes are independent. If $\mathbf{X^2}$ is large there is a high probability the attributes are dependent. Probability verses $\mathbf{X^2}$ tables are published in many statistics textbooks.

A.1.10 Interpolation

Interpolation simply means to calculate a new value in between two known values. Linear interpolation uses the output of a linear regression to arrive at new values of a dependent variable based on the input of an independent variable value in to the linear regression model. That is, interpolation is the creation new values at equal distances along a line between two known values.

A.1.11 Extrapolation

Extrapolation involved projecting out to new values outside the range of known values based on a regression model. In general, extrapolation is not very reliable and the results so obtained are to be viewed with a healthy scepticism. In order for extrapolation to be at all reliable, the original data must be very consistent.

A.1.12 Probability

The probability of an event is the likelihood of it occurring, and is expressed as a number between 0 (certainty that the event will not occur) and 1 (certainty that the event will occur).

In a situation where all outcomes of an experiment are equally likely, then the probability of an event is equal to the number of outcomes corresponding to that event divided by the total number of possible outcomes.

In a deck of 52 playing cards, the probability of drawing a KING would be 4/52, since there are 4 KINGs in the deck. The probability of drawing KINGS or QUEENS would be 8/52. The probability of not drawing a KING or QUEEN would be 44/52.

If A and B represent events, the some properties of probabilities include:

```
Pr(not A) = 1 - Pr(A)
Pr (A or B) = Pr(A) + Pr(B) if A & B are mutually exclusive events
Pr (A or B) = Pr(A) + Pr(B) - Pr(A and B) for any 2 events
Pr (A and B) = Pr(A) x Pr(B) if A and B are independent events
```

A.1.12.1 Conditional probability

Given 2 events A and B, the conditional probability of A given B is the probability that A will occur, given that B has occurred. If B has nonzero probability, then the condition probability of A given B is:

```
Pr(A|B) = Pr(A and B) / Pr(B)
```

A.1.13 Sampling

Critical to statistics is the taking of a sample from a very large population on which to perform analyses. Two important factors that apply to a sample are the following:

1. **Size** of the sample, that is, the number of units selected from the entire population.
2. **Quality** of the sample (or how good or representative is the sample) vis-a-vis the population from which it was extracted.

The selection process is critical to the sample, for example, excluding minorities in a voter survey would significantly taint the results of analysis of such a sample.

A sampling procedure that ensures that all possible samples of 'n' objects are equally likely is called a *Simple Random Sample*. A simple random sample has two properties that make it the standard against which all other methods are measured as follows:

1. Unbiased — each object has the same chance of being chosen.
2. Independence — selection of one object has no influence on the selection of the other objects .

Deriving totally unbiased, independent samples may not be cost effective, and other methods are used to come up with efficient and cost-effective samples. For example, knowing something about the population allows for different techniques such as *Stratified* sampling, *Cluster* sampling and *Systematic* sampling.

A.1.14 Transposition

Transposition is simply moving the rows to the columns and vice versa. This is sometimes called Pivoting.

A.1.15 Histograms

A histogram is a graphical representation of the distribution of a set of data.

A histogram lets us see the shape of a set of data - where its center is, and how far it spreads out on either side. It can provide a graphical representation of other statistics like spread, and skewness.

Skewness describes if the 'tail' is to the left or right. Right-hand skewness is referred to as positive and left-hand is negative.

Individual data points are grouped together into ranges in order to visualize how frequently data in each range occurs within the data set. High bars indicate more data in a given range, and low bars indicate less data. In the histogram shown in Figure A-3, the peak is in the 20-39 range, where there are five points.

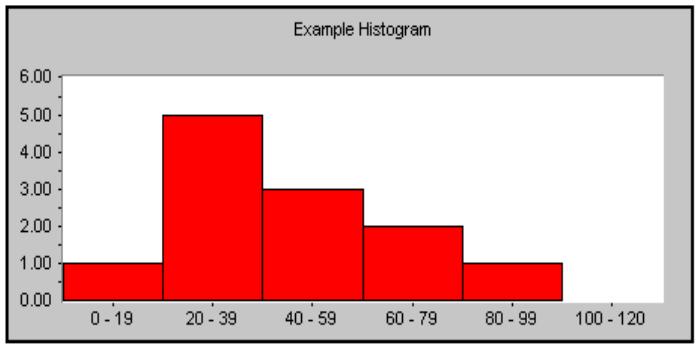


Figure A-3 Histogram

The popularity of a histogram comes from its intuitive easy-to-read picture of the location and variation in a data set. There are, however, two weaknesses of histograms that you should bear in mind:

- Histograms can be manipulated to show different pictures. If too few or too many bars are used, the histogram can be misleading. This is an area which requires some judgment, and perhaps some experimentation, based on the analyst's experience.
- Histograms can also obscure the time differences among data sets. For example, if we looked at data for #births/day in the United States in 1996, you would miss any seasonal variations, e.g. peaks around the times of full moon. Likewise, in quality control, a histogram of a process run tells only one part of a long story. There is a need to keep reviewing the histograms and control charts for consecutive process runs over an extended time to gain useful knowledge about a process.

A.1.15.1 Equi-width histograms

An equi-width histogram is a special case of the above histogram, with the requirement that the x-axis ranges are equally distributed, while the y-axis represents the frequency distribution within each of those x-axis ranges.

A.1.15.2 Equi-height histograms

An equi-height histogram is similar to the typical histogram discussed previously in that it graphically represents the distribution of data, but it uses a slightly different scale. In an equi-height histogram the height of each bar is equal and X-axis range varies.

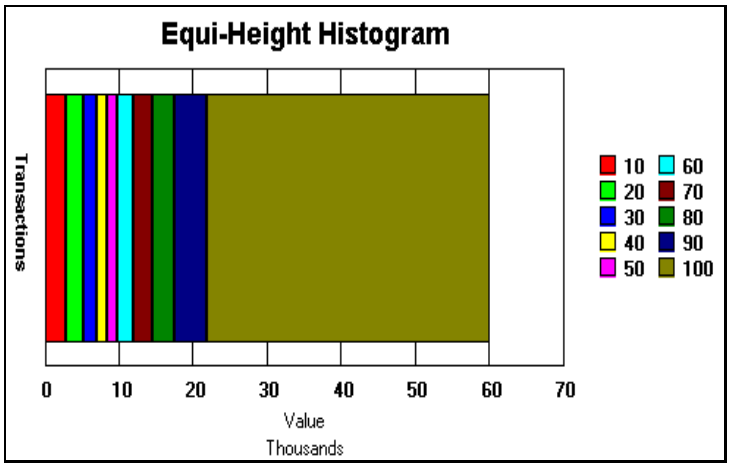


Figure A-4 Equi-height or frequency histogram

B

Tables used in the examples

In this appendix we describe the DDL of the tables used in the examples provided throughout this redbook.

DDL of tables

The tables are presented here in alphabetical order. The content of these tables varied from query to query, and is therefore not shown here.

Example: B-1 AD_CAMP

```
CREATE TABLE “AD_CAMP” (
      “AD_BUDGET” REAL,
      “SALES” REAL)
```

Example: B-2 CAL_AD_CAMP

```
CREATE TABLE “CAL_AD_CAMP” (
      “CITY” VARCHAR(15),
      “AD_BUDGET” REAL,
      “SALES” REAL)
```

Example: B-3 BIG_CHARGES

```
CREATE TABLE "BIG_CHARGES"  (
        "CUSTID" CHAR(10) NOT NULL,
        "CHARGE_AMT" DEC(9,2) NOT NULL)
```

Example: B-4 CUST

```
CREATE TABLE "CUST"  (
        "CUSTID" CHAR(10) NOT NULL,
        "MARITAL_STATUS" CHAR(1),
        "INCOME_RANGE" INTEGER NOT NULL,
        "ZIPCODE" INTEGER,
        "RESIDENCE" VARCHAR(5))
```

Example: B-5 CUST_DATA

```
CREATE TABLE "CUST_DATA"  (
        "CUSTID" INTEGER NOT NULL,
        "PURCHASES" DEC(9,2) NOT NULL)
```

Example: B-6 CUSTTRANS

```
CREATE TABLE "CUSTTRANS"  (
        "CUSTID" CHAR(10) NOT NULL,
        "CHARGE_AMT" DEC(9,2) NOT NULL,
        "DATE" DATE NOT NULL)
```

Example: B-7 EMPLOYEE

```
CREATE TABLE "EMPLOYEE"  (
        "EMPNO" CHAR(6) NOT NULL,
        "FIRSTNME" VARCHAR(12) NOT NULL,
```

```
        "MIDINIT" CHAR(1) NOT NULL,
        "LASTNAME" VARCHAR(15) NOT NULL,
        "WORKDEPT" CHAR(3),
        "PHONENO" CHAR(4),
        "HIREDATE" DATE,
        "JOB" CHAR(8),
        "EDLEVEL" SMALLINT NOT NULL,
        "SEX" CHAR(1),
        "BIRTHDATE" DATE,
        "SALARY" DEC(9,2),
        "BONUS" DEC(9,2),
        "COMM" DEC(9,2))
```

Example: B-8 EVENT

```
CREATE TABLE "EVENT"  (
        "EVENT_NAME" CHAR(15),
        "EVENT_DATE" DATE,
        "ATHLETE" CHAR(8),
        "COUNTRY" CHAR(15),
        "SCORE" DECIMAL(3,1))
```

Example: B-9 FACT_TABLE

```
CREATE TABLE "FACT_TABLE"  (
        "CITY_ID" INTEGER NOT NULL,
        "PRODUCT_KEY" INTEGER NOT NULL,
        "TIME_ID" INTEGER NOT NULL,
        "SCENARIO_ID" INTEGER NOT NULL,
        "TRANSDATE" DATE,
        "SALES" INTEGER,
        "COGS" INTEGER,
        "MARKETING" INTEGER,
        "MISC" INTEGER,
        "PAYROLL" INTEGER,
        "OPENING_INVENTORY" INTEGER,
        "ADDITIONS" INTEGER,
        "ENDING_INVENTORY" INTEGER)
```

Example: B-10 FEB_SALES

```
CREATE TABLE "FEB_SALES"  (
        "CITY" VARCHAR(15),
        "BRANCH_ID" INTEGER,
        "SALES" INTEGER)
```

Example: B-11 LC_PURCHASES

```
CREATE TABLE "LC_PURCHASES" (
```

```
      “CARDNO” CHAR(12) NOT NULL,
      “COFFEE“ DEC(7,2),
      “BEER” DEC(7,2),
      “SNACKS” DEC(7,2),
      “BREAD” DEC(7,2),
    “READY_MEALS” DEC(7,2),
    “MILK” DEC(7,2))
```

Example: B-12 LOC

```
CREATE TABLE “LOC” (
      “LOCID” CHAR(10) NOT NULL,
      “CITY” VARCHAR(10), STATE CHAR(2),
      “COUNTRY” VARCHAR(10))
```

Example: B-13 LOOKUP_MARKET

```
CREATE TABLE "LOOKUP_MARKET"  (
        "REGION" VARCHAR(50),
        "REGION_TYPE_ID" INTEGER,
        "STATE" VARCHAR(50),
        "STATE_TYPE_ID" INTEGER,
        "CITY_ID" INTEGER NOT NULL,
        "CITY" VARCHAR(50),
        "SIZE_ID" INTEGER,
        "POPULATION" INTEGER)
```

Example: B-14 PRICING

```
CREATE “TABLE PRICING” (
      “STORE” CHAR(15) NOT NULL,
      “ITEM” CHAR(10),
      “COST“ DEC(7,2),
      “PRICE” DEC(7,2))
```

Example: B-15 PROD

```
CREATE TABLE "PROD"  (
        "PRODID" CHAR(10) NOT NULL,
        "PROD_TYPE" INTEGER,
        "PROFIT" DECIMAL(5,2),
        "PROD_NAME" CHAR(10))
```

Example: B-16 PROD_OWNED

```
CREATE TABLE "PROD_OWNED"  (
        "PRODID" CHAR(10),
        "PROD_TYPE" INTEGER,
        "OPEN_DATE" DATE,
        "CUSTID" CHAR(10),
```

```
        "BRANCH_ID" INTEGER,
        "CITY" VARCHAR(15))
```

Example: B-17 SALES

```
CREATE TABLE "SALES"  (
        "SALES_DATE" DATE NOT NULL,
        "SALES_PERSON" VARCHAR(15) NOT NULL,
        "REGION" VARCHAR(15) NOT NULL,
        "SALES" INTEGER)
```

Example: B-18 SALES_DTL

```
CREATE TABLE "SALES_DTL"  (
        "CITY" VARCHAR(15),
        "BRANCH_ID" INTEGER,
        "PRODID" CHAR(10),
        "QTY" INTEGER,
        "DATE_SOLD" DATE)
```

Example: B-19 SEEDINGS

```
CREATE TABLE “SEEDINGS” (
      “PLAYER” CHAR(20) NOT NULL,
      “W_RANKING” SMALLINT NOT NULL,
      “T_1” SMALLINT,
      “T_2” SMALLINT,
      “T_3” SMALLINT,
      “T_4” SMALLINT,
      “T_5” SMALLINT)
```

Example: B-20 STOCKTAB

```
CREATE TABLE “STOCKTAB” (
      “DATE” DATE,
      “SYMBOL” CHAR(5),
      “CLOSE_PRICE” DECIMAL(8,3))
```

Example: B-21 SURVEY

```
CREATE TABLE "SURVEY"  (
        "CUSTID" INTEGER,
        "PROD_NAME" VARCHAR(10),
        "CITY" VARCHAR(10))
```

Example: B-22 SURVEY_MORTG

```
CREATE TABLE "SURVEY_MORTG"  (
        "CITY" VARCHAR(10),
```

```
        "BRANCH" INTEGER,
        "MORTG_PREF" INTEGER)
```

Example: B-23 TRAFFIC_DATA

```
CREATE TABLE "TRAFFIC_DATA"  (
        "HITS" INTEGER,
        "DAYS" INTEGER)
```

Example: B-24 T

```
CREATE TABLE “T” (
      “SALES” DEC(9,2),
      “AD_BUDGET” DEC(9,2))
```

Example: B-25 TRANS

```
CREATE TABLE “TRANS” (
      “TRANSID” CHAR(10) NOT NULL,
      “ACCTID” CHAR(10) NOT NULL,
      “PDATE” DATE NOT NULL,
      “STATUS” VARCHAR(15),
      “LOCID” CHAR(10) NOT NULL)
```

Example: B-26 TRANSACTIONS

```
CREATE TABLE “TRANSACTIONS” (
      “STORE” CHAR(15) NOT NULL,
      “QUARTER” CHAR(2)  NOT NULL,
      “ITEM” CHAR(10) NOT NULL,
      “SALES” INT NOT NULL)
```

Example: B-27 TRANSITEM

```
CREATE TABLE “TRANSITEM” (
      “TRANSITEMID” CHAR(10) NOT NULL,
      “TRANSID” CHAR(10) NOT NULL,
      “AMOUNT” DECIMAL(10,2) NOT NULL,
      “PGID” CHAR(10) NOT NULL)
```

C

Materialized view syntax elements

In this appendix we describe the main syntax elements associated with creating and refreshing materialized views.

Materialized view main syntax elements

Figure C-1 describes the main syntax elements for creating and refreshing materialized views.

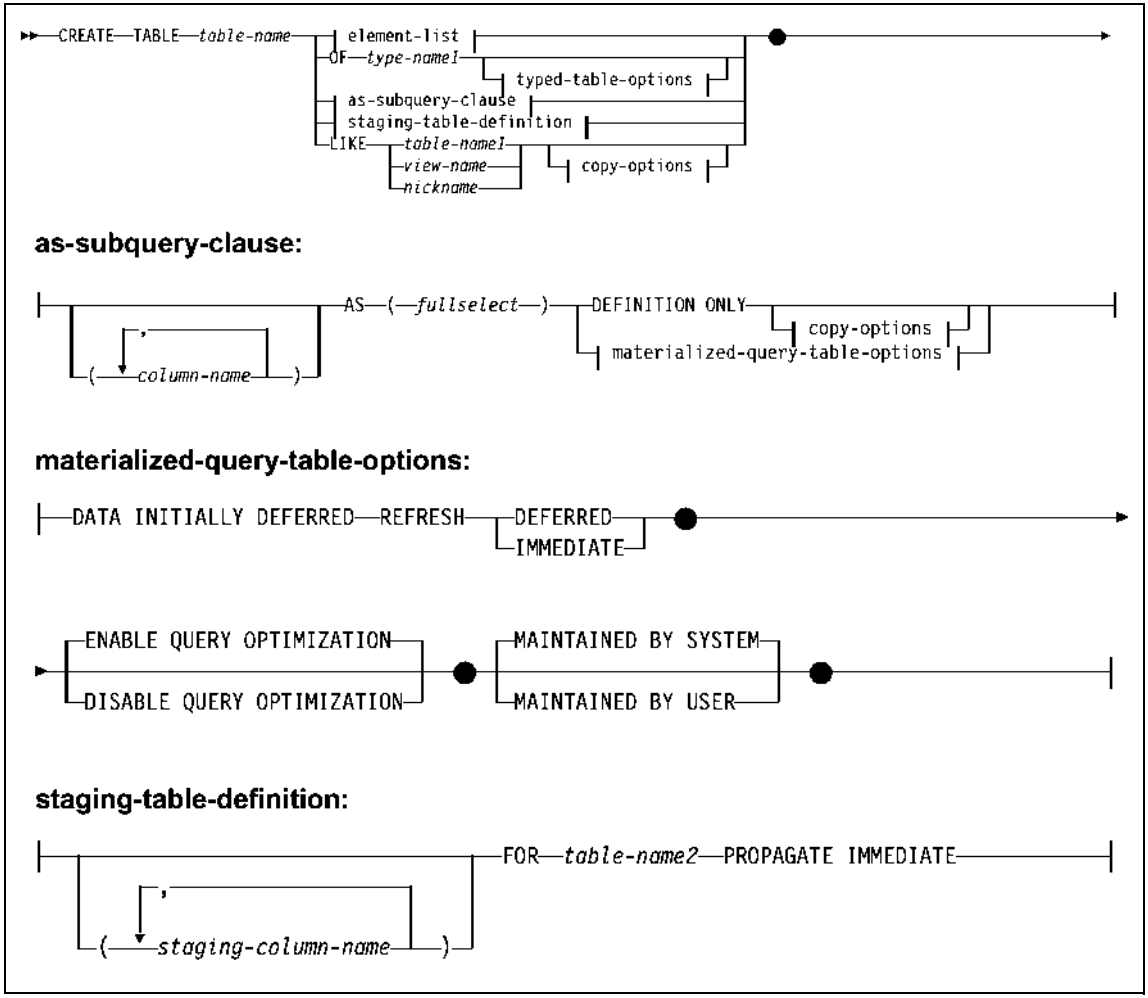


Figure C-1 Main syntax elements of materialized views

When DEFINITION ONLY is specified, any valid fullselect that does not reference a typed table or typed view can be specified.The query is used only to define the table. Such a table is *not* a materialized view. Therefore, the REFRESH TABLE statement cannot be used.

With the DATA INITIALLY DEFERRED option, data is not inserted into the table as part of the CREATE TABLE statement. The materialized view has to be populated using the SET INTEGRITY command or a REFRESH TABLE statement, or some other user determined mechanisms depending upon whether the materialized view is system maintained or user maintained.

The ENABLE QUERY OPTIMIZATION parameter allows the materialized view to be used for query optimization.

The DISABLE QUERY OPTIMIZATION ensures that the materialized view is not used for query optimization, however, it can still be directly queried

The MAINTAINED BY SYSTEM option indicates that the data in the materialized view is maintained by the system and it is the default.

The MAINTAINED BY USER option indicates that the materialized view is maintained by the user. The user is allowed to perform update, delete, or insert operations against the user-maintained materialized view. The REFRESH TABLE statement, used for system-maintained materialized views can **not** be invoked against user-maintained materialized views. Only a REFRESH DEFERRED materialized view can be defined as MAINTAINED BY USER.

Figure C-2 shows the syntax of the REFRESH statement that refreshes the data in a materialized view.

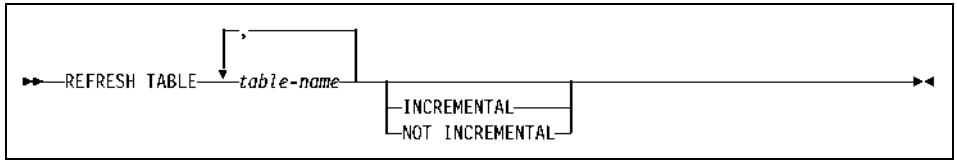


Figure C-2 REFRESH TABLE statement

These are the options for the REFRESH statement:

- The INCREMENTAL option specifies an incremental refresh for the table by considering only the appended portion (if any) of its base tables, or the content of an associated staging table (if one exists, and its contents are consistent).
- The NOT INCREMENTAL option specifies a full refresh for the table by recomputing the materialized view definition.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 246.

- *Business Intelligence Certification Guide, SG24-5747*

Other resources

These publications are also relevant as further information sources:

- *Answering Complex SQL Queries Using Automatic Summary Tables, IBM Almaden Research Center, Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, Monica Urata*, which may be obtained from the Technical Library in the IBM Almaden Research Center, San Jose, CA.
- *IBM DB2 UDB SQL Reference*, SC09-4845
- *DataBase Associates: IBM Enterprise Analytics for the Intelligent e-business*:

  `http://www-3.ibm.com/software/data/pubs/papers/bi/bi.pdf`
- *Sampling Techniques by Cochran, Wiley, 1977*, ISBN 0-461-16240-X
- *The Cartoon Guide to Statistics, Gonick & Smith, HarperPerenial, 1993*, ISBN 0-06-273102-5
- *Forgotten Statistics, Downing & Clark, Barronn’s, 1996*, ISBN 0-8120-9713-0
- *Statistics for the Utterly Confused, Lloyd Jaisingh, McGraw Hill, 2000*, ISBN 0-07-135005-5

Referenced Web sites

These Web sites are also relevant as further information sources:

- IBM DB2 UDB Technical Library

  `http://www-4.ibm.com/cgi-bin/db2www/data/db2/udb/winos2unix/support/v7pubs.d2w/en_main`

How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

ibm.com/redbooks

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Index

F

G

H

I

J

L

M

N

O

P

T

U

V

W

Z

DB2 UDB's High-Function Business Intelligence in e-business

(0.5" spine)
0.475"<->0.875"
250 <-> 459 pages

DB2 UDB's High-Function Business Intelligence in e-business

Exploit DB2's materialized views (ASTs/MQTs)

Leverage DB2's statistics, analytic and OLAP functions

Review sample business scenarios

This IBM Redbook deals with exploiting DB2 UDB's materialized views (also known as ASTs/MQTs), statistics, analytic, and OLAP functions in e-business applications to achieve superior performance and scalability. This redbook is aimed at a target audience of DB2 UDB application developers, database administrators (DBAs), and independent software vendors (ISVs).

We provide an overview of DB2 UDB's materialized views implementation, as well as guidelines for creating and tuning them for optimal performance.

We introduce key statistics, analytic, and OLAP functions, and describe their corresponding implementation in DB2 UDB with usage examples.

Finally, we describe typical business level queries that can be answered using DB2 UDB's statistics, analytic, and OLAP functions. These business queries are categorized by industry, and describe the steps involved in resolving the query, with sample SQL and visualization of results.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-6546-00 ISBN 0738424609