READY: Completeness is in the Eye of the Beholder

Badrish Chandramouli [†] Johannes Gehrke [†] Jonathan Goldstein [†] Moritz Hoffmann [‡]
Donald Kossmann [†] Justin Levandoski [†] Renato Marroquin [‡] Wenlei Xie [◇]
Microsoft Corp., Redmond, USA[†]
{badrishc, johannes, jongold, donaldk, justinle}@microsoft.com
Systems Group, ETH Zurich[‡] Facebook Inc., Menlo Park, USA [◇]
{marenato, moritzho}@inf.ethz.ch wxie@fb.com

ABSTRACT

Modern database systems support one set of integrity constraints per database. Imagine you could specify multiple sets of integrity constraints per database, one for each type of application. This paper argues why this might be a good idea and introduces a system that implements this idea.

1. INTRODUCTION

Data lakes in the cloud have proliferated in the last years, driven by suitable infrastructure support which has been added by all major cloud providers. In data lakes, diverse applications share data and associated computational resources to process and query the data. However, with great capabilities often comes great peril unless data and resource governance creates order from chaos. In this paper, we consider a special type of data governance, namely the data integrity constraints that are needed by applications. A traditional OLTP system or data warehouse comes with a carefully crafted set of integrity constraints – not too stringent to be prohibitive, but not too relaxed such that applications need to consider all corner cases. In a data lake where we have a multitude of applications accessing shared data, such a shared set of integrity constraints is hard to find. If we are too stringent, we prevent many applications that require relaxed constraints from running, but if we are too relaxed, the data lake becomes the greatest common denominator of all constraints and then we have to rely on the checking of integrity constraints and associated corner cases in application code. Thus, we need a system that enables the creation of different sandboxes within a data lake, each sandbox with its associated set of integrity constraints.

Consider an example of a data lake of products and orders with applications that have very different integrity constraints:

- A stock-keeping application that runs reports every hour and aggregates the orders for all products. We only want to display a report for a product once all the orders for the hour have been completed.
- A dashboard application that reports on the total value of orders by product and the total value of orders by customers, receiving as input feeds of new orders and feeds of new customers. If orders are ahead of associated customers, the dashboard would generate inconsistent results.
- Another dashboard application that shows completed orders that have shipped. This means that only orders where all lineitems have shipped should be displayed.

All these application query the same data lake, but they have very different requirements on the integrity of states of the data that they want to see.

Contributions of this Paper. In this paper, we introduce a new concept for these types of applications: A data lake with multiple sandboxes with different integrity constraints. Each application group gets its own sandbox with its own integrity constraints that protects the application from seeing inconsistent states according to its own integrity constraints on the database. Multiple applications that have the same requirements can run in the same sandbox. All of this should happen transparently to the developers of the applications. We first introduce the concept of a data lake with multiple integrity constraints, describe design goals, the design space, and how the concept relates to and differs from previous work (Section 2). We then describe an instance of a data lake with multiple integrity constraints, the READY System, which efficiently handles a novel type of integrity constraint that we call a completeness constraint. We describe the nature of such constraints and associated extensions to SQL and sketch the implementation of our first prototype (Sections 3 and 4). We then detail the results of a preliminary evaluation of READY as compared to two natural baselines, and we show that we can modify existing mechanisms for handling temporal data to implement data lakes with multiple constraints (Section 5). We discuss related work in Section 6, and we conclude with a discussion of future work in Section 7.

2. DATA LAKES WITH INTEGRITY CONSTRAINTS

This section describes the main concepts of our approach to add integrity constraints to a data lake and discusses how these concepts relate to traditional database systems.

2.1 Goals

By adding integrity constraints to a data lake, we would like to achieve three goals:

- *Sharing:* There is conceptually only one copy of data shared across many applications and users. Updates to data are potentially immediately visible to all applications without copying the data.
- *Custom Integrity:* Each application defines its own integrity constraints and it only sees data and states of the data lake

8th Biennial Conference on Innovative Data Systems Research (CIDR '17)
January 8-11, 2017 , Chaminade, California, USA.

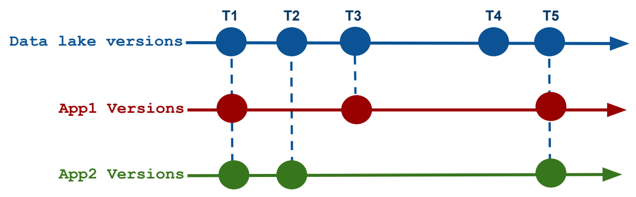

Figure 1: Temporal Data Lake

that comply with its integrity constraints.

- *Decoupling:* Applications do not block each other. Each application makes progress independently of the integrity constraints defined for other applications.

These three goals are in conflict to each other. It is easy to have two of these three, but not all three. For instance, applications are *decoupled* with *custom integrity* today by maintaining a separate copy of the database for each application; i.e., data silos which sacrifice sharing. It is also possible to support *sharing* and *decoupling* without any integrity constraints; that is the status quo of many data lakes today. Finally, it is also possible to have *sharing* and *custom integrity* (and no decoupling); in this scenario, the database must comply to the *union* of all integrity constraints of all applications which blocks applications that only require compliance to a subset of the integrity constraints. This configuration is often implemented in a data warehouse with an ETL process which might *over-regulate* data quality to be on the safe side.

As shown in Section 3, READY achieves these three goals by supporting a specific scenario only: information flows between producers and consumers which are particularly common in data warehousing scenarios. The underlying concepts of READY, however, are more general and can be applied in different ways to support other scenarios, too. The remainder of this section describes these general concepts.

2.2 Sandboxes and Data Lake Versions

The main idea is that each application (or class of applications) runs in a sandbox. A sandbox specifies the specific integrity constraints of that application. Applications that share the same set of integrity constraints can run in the same sandbox. Sandboxes are, thus, the key concept that implements the *custom integrity* goal.

Figure 1 shows how we propose to implement *sharing* and *decoupling*. Figure 1 depicts three timelines. The first (blue) timeline shows five different versions of the data lake at times $T_1, \ldots, T_5$. For instance, at T_1 the data lake could be empty. At T_2, a file with new orders is uploaded to the data lake. At T_3, some of these orders are updated. And so on. So, the first extension we propose is to view the data lake as a versioned data lake in which each update, insert, or delete creates a new version of the data lake.

A sandbox controls which versions of the data lake are *visible* to an application that runs in the sandbox. A version of the data lake is visible if it meets the integrity constraints specified in the sandbox. For instance, Versions T_1, T_2, and T_5 of the data lake meet the integrity constraints of the sandbox of *App1* in Figure 1. Versions T_2 and T_4 are not visible.

In addition to sandboxes, we propose to define a transaction-specific policy that determines which visible version of the data lake is used to execute a transaction. At Time T_2, for instance, such a policy would determine whether a query issued by *App1* is executed using Version T_1 or whether *App1* waits for Version T_3 in order to execute the query. There can be many different policies on how to select the right visible version at every point in time, and Section 3.3 describes the options we chose to implement for the first READY prototype.

In summary, sandboxes control which versions of a data lake are visible to an application. This way, sandboxes implement *custom integrity* by guaranteeing that applications only run on versions of the data lake that meet their integrity constraints. Furthermore, the applications are totally *decoupled*. There might be overlap (e.g., Versions T_1 and T_5 are visible to both applications in Figure 1), but every application defines its own sandbox and thus defines its own visibility. Furthermore, no application is ever blocked by the constraints of another application. It is possible to apply the updates to generate Version T_4 of the data lake even though Version T_4 does not meet the constraints of neither application. Finally, there is still only one copy of each object in the data lake, thereby enabling *sharing*. For instance, all updates (e.g., insertions of new orders) made at T_5 are immediately visible to both *App1* and *App2* without doing an extra copy of these objects.

2.3 Discussion of Related Concepts

How do these concepts relate to concepts in traditional database systems? The closest analogy of a sandbox is a *view* or more precisely a *temporal view*. For instance, we could encode *App1*'s integrity constraints into a view definition and issue all of *App1*'s queries against this view. This way, *App1* would *see* Version T_1 of the data lake at Time T_2. There are three reasons why views are not a good way to add integrity constraints to data lakes. First, the view definition that includes integrity constraints would be humongously complex. In contrast, the sandbox definition can be specified in a straight-forward way, just as integrity constraints in SQL today. (Section 3.2 contains examples.) Second, temporal views are not updatable. As we will see, however, we also want to run update transactions in sandboxes and make sure that they meet the integrity constraints of the updating application. Third, (temporal) views can look into the past at best. However, at Time T_2 it might serve *App1* best to wait for Version T_3 to continue processing queries.

The second related concept of traditional databases are *integrity constraints*. In fact, we completely adopted this concept. A sandbox is nothing else than a container for a set of integrity constraints. We need the notion of a sandbox only because we want to support *multiple* sets of integrity constraints in a data lake, one set per class of applications in order to *decouple* the applications that run in the data lake.

Finally, versioning the data lake fits nicely into the concept of a temporal database [14]. In fact, READY 1.0 is implemented that way (Section 4). However, the concepts are more general and it is not necessary to implement versioning or a temporal database. For instance, if *App1* waits at Time T_2 for the next consistent version (i.e., T_3), then there is no need to keep Version T_1 of the data lake.

3. READY 1.0

The sandbox model sketched in Section 2 is generic and there are many different ways to instantiate it. This section describes the specific approach we took in our first implementation of the READY system. This approach is specifically geared towards a particular application pattern: controlling the information flow between producers and consumers. This pattern is particularly common in data warehousing with an OLTP system as a producer and data marts as consumers. For instance, the producer could be an ERP system that generates new orders and updates these orders as they are fulfilled. The consumers could be different materialized views that analyze the orders in order to make business decisions; e.g., launch marketing campaigns or replenish supplies. The producer/consumer pattern also supports real-time dashboards which are updated whenever a new (consistent and complete) batch of events have been processed or the generation of periodic reports for decision support.

3.1 Producers and Consumers

READY 1.0 differentiates between two kinds of applications and users: producers and consumers. Producers carry out read and update transactions against the data lake. That is, they transform the data lake from one state into the next state. Consumers carry out read-only transactions on different versions of the data lake.

The transactions of both producers and consumers run in sandboxes which specify the specific integrity constraints of the producers and consumers. For instance, the marketing department may want to wait until all the orders of a particular quarter have been completed (shipped or canceled) before making decisions on how to promote a particular product in a certain region. In contrast, replenishment decisions are made on a weekly or daily basis and orders are processed by the ERP system in real-time.

Furthermore, READY 1.0 models the data lake as a sequence of versions as shown in the example of Figure 1. In READY 1.0, there is exactly one producer sandbox and all update transactions that generate new versions of the data lake must run in this producer sandbox. This producer sandbox specifies the constraints that any version of the data lake must fulfill and update transactions run in this producer sandbox in the same way as update transactions run in a traditional database with only one set of integrity constraints. If the data lake does not meet the constraints of the producer sandbox at the end of an update transaction, then the commit of the transaction fails and all the updates are rolled back; again, just like in any other traditional database system. In the example of Figure 1, the first (blue) timeline represents the producer.

Note that the restriction to one producer sandbox does *not* restrict to having only one data source. The data lake could, for instance, be populated by an ERP system that uploads new orders and a CRM system that generates new customers. However, these different systems (or the ETL process that imports data from these systems into the data lake) need to agree on a single set of integrity constraints which is captured in the producer sandbox.

For a (read-only) consumer, the sandbox defines which versions of the database the consumer sees. In READY 1.0, there can be zero, one, or any number of consumer sandboxes. For instance, *App1* and *App2* are consumers in Figure 1, each running in their own sandbox.

3.2 Sandbox Syntax and Completeness Constraints

The syntax to define a sandbox is as follows:

```
CREATE SANDBOX sandboxName ( argname argtype )*
[ FOR UPDATES ]
[ WHEN predicate ]
[ WITH ( relationName: predicate )* ];
```

Listing 1: Sandbox Syntax

The optional `FOR UPDATES` clause indicates the producer sandbox. READY 1.0 only supports one producer sandbox and returns an error if an application tries to create a second producer sandbox with a `FOR UPDATES` clause.

The `WHEN` clause contains a predicate that involves all the integrity constraints, possibly as a conjunction of predicates. Only versions of the data lake that comply with the *WHEN predicate* a visible to queries and transactions running in that sandbox. For instance, the following sandbox definition indicates a sandbox that makes sure that applications do not see any orders whose status is "open":

```
CREATE SANDBOX noOpenOrderSandabox
WHEN NOT EXISTS
  (SELECT * FROM Order WHERE status = "open");
```

Listing 2: Sandbox Example

One particularly common class of constraints for producer / consumer scenarios is *completeness*. For instance, the marketing departments wants to make decisions only when *all* orders of a quarter have been processed. As another example, we may want to analyze the telemetry of a rack of machines in a data center only when *all* machines of the rack have reported their status. Or, we want to ship the next version of a software product once *all* unit tests have passed. Unfortunately, the SQL syntax does not support *universal quantification* which is the basis to specify *completeness* constraints. READY, however, does support *FORALL* predicates in the `WHEN` clause to make it easier to specify completeness predicates.

Sandboxes can be parameterized. For instance, an analyst might be interested in orders of customers from a particular country only and, thus, specify a *WHEN predicate* that involves those customers only. The following parameterized sandbox definition makes sure that such analysts only see versions of the data lake in which all the orders of such customers have status "Verified".

```
CREATE SANDBOX completeByNation(:nationId INT)
WHEN
  FORALL (SELECT o.status as s
          FROM Order o, Customer c
          WHERE o.o_custkey == c.c_custkey
            AND c.c_nationkey == :nationId)
  SATISFY s = "Verified"
```

Listing 3: Parameterized Sandbox

In addition to a `WHEN` clause which specifies which versions of a data lake are visible, a sandbox can also contain the definition of one or several `WITH` clauses. A `WITH` clause defines which tuples of a relation are visible. For instance, such a `WITH` clause may specify that an application running in the data lake only sees the telemetry of machines of a data center that have reported load statistics in the last 5 seconds. We call this concept *fine-grained integrity constraints*. The visibility of records defined by the `WITH` clause of a sandbox is orthogonal to the visibility of entire data lake versions specified by the `WHEN` clause. We mention `WITH` clauses because READY supports them, but we will not discuss `WITH` clauses in the remainder of this paper which is focused on integrity constraints defined in `WHEN` clauses.

3.3 Sandbox Usage and Queries

In READY 1.0, all transactions run in a sandbox. The transactions of producers always run in the one and only producer sandbox. By definition, this sandbox is always compliant with the latest, current version of the data lake so that all updates are applied to the latest version of the data lake. The queries of consumers, however, may run on different versions of the data lake. To specify the version, READY extends the SQL syntax for transaction and introduces an additional keyword which can take one of the following three values:

- *LAST:* The query is evaluated using the last consistent state of the database, possibly the current state of the database. Such a state exists because initially, the empty database is consistent according to all sandboxes. *LAST* is also the default if nothing else is specified and it is the mandatory mode for the usage of a producer sandbox.

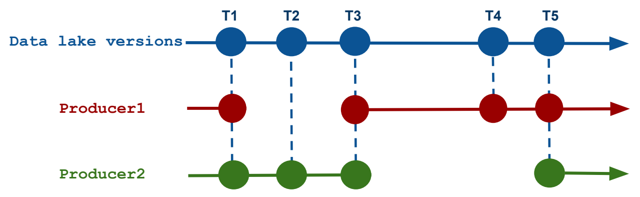

Figure 2: Collaboration Scenario

- *NEXT:* If the current state of the database is not consistent with regard to the constraints of the sandbox, then the query blocks until the database is in a consistent state. At that point (in future), the query is evaluated and the results returned to the consumer. If the current state of the database is consistent, *LAST* and *NEXT* are identical.

- *CONTINUOUS:* The query is applied to all consistent states of the database, now and in the future. The consumer must register a callback function to consume the results of the continuous READY query. Continuous queries are useful in many reporting applications; for instance, an enterprise may want to report on the business activity at the end of every business day once all orders of that day have been fully processed.

The following is an example query that runs in the parameterized sandbox of Listing 3. The query asks for aggregates of *lineitems* of customers of a particular country. Since it runs in the *completeByNation* sandbox for "Germany", the query blocks until the data lake has reached a state in which all orders of German customers have been verified.

```
BEGIN USING completeByNation("Germany") NEXT;
  SELECT c.name, count(*)
  FROM Order o, Customer c
  WHERE o.o_custkey = c.c_custkey
    AND c.c_nationkey = "Germany"
  GROUP BY c.name
COMMIT;
```

Listing 4: Using a Sandbox

Any number of SQL statements can be executed within the `BEGIN USING` and `COMMIT` bracket. In READY 1.0, the statements of a transaction run in the same sandbox and on a single version of the data lake. Therefore, the `BEGIN USING` and `COMMIT` bracket also defines the boundaries of a transaction in READY 1.0.

3.4 Outlook: Multiple Producer Sandboxes

READY 1.0 supports producer / consumer scenarios which are typical for many business scenarios today (e.g., ERP and decision support applications). The concepts of Section 2 can also be used to support more complex collaborations in which applications can run read and update transactions through multiple producer sandboxes that each have their own set of integrity constraints.

Figure 2 depicts such a scenario. Again, Figure 2 shows three timelines. The first (blue) timeline depicts five versions of the data lake. This time, these versions are generated by transactions created by two producers. Producer 2 generates Versions T_2 and T_3 of the data lake. Producer 1 generates Versions T_4 and T_5. Again, the sandboxes control which versions are visible to each application. Producer 1 sees Versions T_1, T_3, T_4, and T_5. Producer 2 sees Versions T_1, T_2, T_3, and T_5.

In this scenario (just as in the simple producer / consumer scenario implemented in READY 1.0), there is a linear sequence of versions of the data lake and each update transaction of a producer operates on the latest, current version of the data lake and generates the next version of the data lake. What makes this "multiple producer sandbox" scenario special is that not every producer sees all versions of the data lake. For instance, Producer 1 does not see Version T_2 of the data lake and is, thus, blocked at this point of time. That is, all update transactions of Producer 1 fail when issued at Time T_2 and Producer 1 needs to wait until Producer 2 is done with its changes and has created a version of the data lake that is consistent with Producer 1's requirements (Version T_3 in this example). The approach depicted in Figure 2, thus, sacrifices *decoupling* of applications for a richer collaboration scenario with more *custom integrity* for each producer.

In general, it is not even necessary that the versions of the data lake are ordered by time as in the examples of Figures 1 and 2. For instance, the model also supports *branching* in which versions of different branches are not ordered. git and other software version control systems are example systems that support branching to effect collaboration. Branching sacrifices *sharing* (i.e., immediate visibility) for *decoupling*. All these examples demonstrate that there are many ways to instantiate the generic sandbox concepts of Section 2. In all cases, it is possible to define sandboxes which control the visibility of versions of the data lake.

4. READY PROTOTYPE

To study the trade-offs of the READY concepts, we built a prototype of READY 1.0 on top of Spark. This prototype is particularly geared towards an information flow pattern that matches data warehousing applications. This section describes this prototype. Section 5 discusses the results of the performance experiments conducted using this prototype.

Figure 3 gives an overview of the building blocks of the READY 1.0 prototype. The application layer is depicted in red. It consists of the OLTP systems that generate data and the analytics and reporting tasks that consume the data. The sandbox definitions for the producer and all the consumers are also part of the application layer. The implementation of the data lake (i.e, Spark) is depicted in green. We use HDFS as a storage layer. Furthermore, we use the Spark SQL processor and Map Reduce engine to implement all READY components which are depicted in green: (a) *Data Ingest* which batches updates and stores them in HDFS; (b) *Version Selection* which determines which version of the (versioned) data lake meets the constraints of a sandbox; (c) *Temporal Query Processor* which rewrites a SQL query issued by a consumer and executes it on the right version of the data lake, thereby using the data lake's SQL processor. We sketch the design of these three green components in the remainder of this section.

4.1 Data Ingest: Storage Layer

The *Data Ingest* component batches a stream of updates (inserts, updates, and deletes) of records and stores them as files in HDFS. READY differentiates between two kinds of files: (1) *data files* which are created as part of *inserts* and contain a set of new records and (2) *update files* which contain updates and deletes and are applied to the records of data files to create new versions of the data files.

Figure 4 gives an example that shows how READY organizes these data and update files for versioned, temporal data management. Data files are depicted in blue, update files are shown in brown. The figure shows a timeline of four versions of the data lake. At the beginning, Version V1, there is only one data file in the

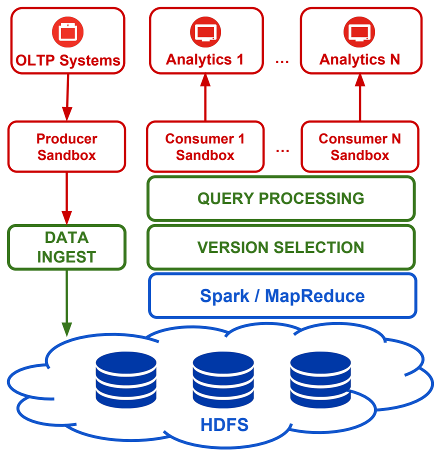

Figure 3: Overview of READY Prototype

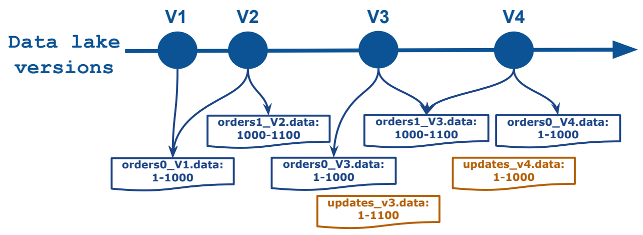

Figure 4: READY Version Management

data lake, “orders0_V1.data”. This file contains the first batch of orders uploaded to the data lake. Version V2 is created by uploading a new batch of orders: The orders are stored in the second data file, “orders1_V2.data”. The third version, Version V3, is created by applying a batch of updates from update file “updates_V3.data”. As a result, READY creates two new data files “orders0_V3.data” and “orders1_V3’.data” which become visible at V3 of the data lake instead of the original data files “orders0_V1.data” and “orders1_V2.data”. Version V4 is created by applying the updates of “updates_V4.data”. These updates only involve updates to orders of “orders0_V3.data”. As a result, only a new version of that file needs to be created and files “orders0_V4.data” and “orders1_V3” are visible in V4 of the data lake.

To implement the visibility of data files for each version of the data lake, READY implements a *VersionMap* that associates each version of the data lake to a set of data files. The *VersionMap* data structure is depicted by the arrows from versions to files in Figure 4.

Figure 5 shows in more detail how READY applies update file “updates_V3.data” to the two data files to generate the new data files of Version V3. We call this process *Delta Materialization*. In Figure 5, READY creates “orders0_V3.data” by applying the updates of “updates_V3.data” to Orders 1 and 7. Likewise, READY creates “orders1_V3.data” by applying the update to Order 15. Applying bulk updates in this way is like processing a join [5] between the existing data files and the update batch. READY performs this join by sorting the records in the update file and then applying them block-wise to the data files. This way, READY reads each data file that is involved in the batch of updates only once. Listing 1 shows the pseudo-code for Delta Materialization.

One reason why READY adopted Delta Materialization and updates all records in the granularity of files is that HDFS only supports updates in the granularity of files: HDFS does not support “update-in-place” in a file so there is no way to update, say Order 1 in File “orders0_V1.dat” without changing the rest of that file. The second reason is that such coarse-grained updates are sufficient in many data warehousing applications because only a few data files are involved in a batch of updates. The reason is that there is a great deal of temporal locality in these applications: Records such as *orders* are typically updated in bursts and often only in a short time period after they were created. For applications that involve small and randomly scattered updates, however, the “file-grained” implementation of updates of our READY prototype might not be appropriate.

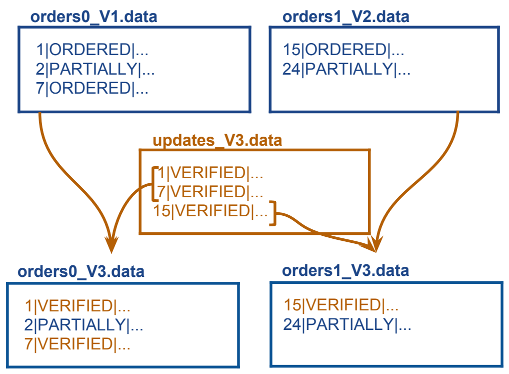

Figure 5: Delta Materialization Example

To increase the effectiveness of Delta Materialization and have “update” batches that are as large as possible, Delta Materialization is carried out in a lazy way. The data files of a version of the data lake are materialized only if needed for query processing. That is, READY’s data ingest component collects updates in a “pending update file” (such as “update_V3.data”) as long as they are not visible in any sandbox and not queried.

Algorithm 1 Delta Materialization

```
1: Input $updBlk$ : block containing updated tuples
2: Input $relationIndex$ : Range index mapping key ranges to relation blocks
3: method Process_Updates($updBlk$, $relationIndex$)
4:     for all rows $r \in updBlk$ do:
5:         $blkRange \leftarrow relationIndex.getRange(r.key)$
6:         Load the relation blocks in $blkRange$
7:         Update and write updated blocks to new location
8: End Method
```

Once Delta Materialization is done, READY can immediately garbage collect the update file. For instance, READY deletes file “updates_V3.data” after it generated files “orders0_V3.data” and “orders1_V3.data”. Garbage collection of data files depends on the visibility of versions of the data lake in sandboxes. If no sandbox needs V1 and V2 of the data lake anymore (the latest compliant version is V3 or later for all sandboxes), then READY can garbage collect the data files “orders0_V1.data” and “orders1_V2.data”. To this end, READY maintains a *watermark* of the oldest relevant version of the data lake for every sandbox and adjusts this watermark whenever it carries out Version Selection for that sandbox. How to do Version Selection is the subject of the next subsection.

4.2 Version Selection

The second important component of our READY prototype is *Version Selection* (Figure 3). Version Selection determines which version of the data lake complies to which sandbox.

Version Selection involves evaluating the integrity constraints of a sandbox, the `WHEN` clause. As shown in Section 5, this evaluation is expensive and is worth optimizing. We have implemented two variants of Version Selection: (1) Eager and (2) Lazy. The Eager variant checks the constraints of every sandbox for every new version of the data lake. More precisely, Eager checks the constraints of *all* whenever READY carries out Delta Materialization to execute the query in *any* sandbox. In contrast, the Lazy variant only checks the constraints of a sandbox when a query is processed in that specific sandbox.

As will become clear, the trade-offs of the Eager and Lazy approaches are straightforward. Eager does more frequent constraint checking (possibly unnecessary work if a sandbox is never used), but it can use more efficient algorithms because it can check constraints *incrementally*. Furthermore, Eager is the more general approach because it can be applied to any query, LAST, NEXT, and CONTINUOUS. Lazy, in contrast, can only be applied effectively to LAST queries.

4.2.1 Eager, Incremental Constraint Checking

Eager constraint checking checks the `WHEN` clause of *all* sandboxes for all (materialized) versions of the data lake. The most efficient way to do that is to consider the predicate of the `WHEN` clause of the sandbox as a continuous query or materialized view and then to update this materialized view incrementally using well-known techniques for incremental updates of materialized views; e.g., [3, 6, 16].

In most READY applications, the predicates in the `WHEN` clauses are either completeness predicates (*forall*) or existential predicates (*exists*). These predicates can be best evaluated incrementally by using a counter. To evaluate a completeness predicate that checks that, say, all orders have status *verified*, we incrementally count all orders and all orders with status *verified*. If the two counters have the same value, we know that we are complete. Likewise, existential predicates can be implemented with a counter; in this case, we are *ready* if the counter is greater than 0. These ideas have been studied extensively in the past and, again, we refer the interested reader to the literature for details (e.g., [3, 6, 16]).

When a new sandbox is created, READY needs to initialize the counters, or more generally the materialized view for the `WHEN` clause. READY does that using the current version of the data lake which may or may not meet the integrity constraints of the new sandbox.

4.2.2 Lazy, From Scratch Constraint Checking

The Eager variant is the only option for CONTINUOUS and NEXT queries. If most queries are LAST queries and sandboxes are used in a sparse way, the Eager variant might be wasteful by unnecessarily checking every version of the data lake for every sandbox. The lazy variant in contrast, does not implement the `WHEN` clause as a materialized view and, thus, does not maintain this view incrementally. Instead, it tries to find the most recent version of the data lake for a given sandbox whenever a query is processed in that sandbox.

READY implements the Lazy approach using a two-phase algorithm. The first phase is done with a Map/Reduce job which groups and sorts all records by *version*. The second phase sweeps through the versions in order to find the latest version that meets the integrity constraints of the sandbox. In order to speed up the *sweep* and contain the search, each sandbox *caches* the timestamp of the last known consistent version of the data lake.

4.3 Query Processing

Once we have created a versioned data lake and carried out Version Selection for all sandboxes and all versions of the data lake, transaction and query processing in a sandbox is straightforward: Depending on the *NEXT/LAST/CONTINUOUS* annotation of the transaction, READY selects the right version of the data lake and then executes the query on all the files visible in that version of the data lake, thereby using the *VersionMap*. If we would like to issue a query with *LAST* annotation at Time V2 in a sandbox whose integrity constraints are fulfilled for V1 and V3 in Figure 4, then READY would use Version V1 because V2 is not visible to that sandbox. Correspondingly, READY would use the *VersionMap* to execute the query on file "orders0_V1.data". If the query has a *NEXT* annotation, READY would block and wait for V3 and then execute the query on files "orders0_V3.data" and "orders1_V3.data".

5. PRELIMINARY EXPERIMENTS

This section presents the results of preliminary experiments using an extension of the TPC-H benchmark and our READY prototype. We compare READY with two other ways to implement integrity constraints in a data lake.

5.1 Systems Under Test

In addition to READY, we studied two other approaches as baselines:

- *Global:* This approach simulates a unified data warehouse for all consumers. It involves two (non-versioned) copies of all data and an ETL process: One copy for the producer (e.g., an OLTP system) and one copy for all consumers (i.e., the data warehouse). The ETL process controls which data (inserts and updates) are copied from the producer system to the consumer system. Only if all the integrity constraints of all consumers are met (i.e., the conjunction), the ETL process copies data from the producer system to the consumer system.

- *Personal:* For N different consumers with different integrity constraints, this approach involves $N + 1$ copies of the data and N ETL processes. That is, there is one copy of the data for the producer and one copy of the data for each consumer. This approach corresponds to the construction of specialized data marts which is common in data warehouses. Furthermore, there is one ETL process for each of the N classes of consumers, making sure that the integrity constraints of that class of consumers are met.

As we will see, the *Personal* approach is more expensive because it keeps more copies of the data, but all data marts are fresh at all times. Overall, Personal scores high in the *customer integrity* and *decoupling* goals, but low in *sharing* (Section 2). In contrast, the *Global* approach loses in terms of data freshness because data only becomes available to a consumer when it meets the requirements of all other consumers, too. In other words, Global scores high on *sharing* and *custom integrity*, but not on *decoupling*. READY tries to score high on all three goals.

We implemented Personal and Global on top of Spark using the same techniques as for READY as much as possible. In particular, we used Delta Materialization and incremental constraint checking for the ETL processes of Personal and Global (Sections 4.1 and

4.2). The only difference is that Personal and Global do not need to maintain a versioned data lake: Conceptually, Personal and Global can do *update in place*, whereas READY needs to implement updates by creating a new version. In our implementation of Delta Materialization, an *update in place* means that Personal and Global can garbage collect an updated data file immediately (resulting in lower storage costs) whereas READY needs to keep old versions of data files until they are guaranteed to be no longer relevant for any sandbox.

5.2 Benchmark

We used the database generator, queries, and refresh functions of the TPC-H benchmark. All experiments reported in this paper were conducted using TPC-H databases with scaling factors 1 and 10 (1 GB and 10 GB of raw data). The refresh functions (all inserts, updates, and deletes) are executed in a (single) producer sandbox which allows any data lake state; i.e., an empty `WHEN` clause. The queries run in a varying number of consumer sandboxes with different sets of integrity constraints.

To simulate the compliance of different data lake versions to various consumer sandboxes, we extended the TPC-H benchmark in the following way. In the original TPC-H benchmark, an order can take one of three states: `Ordered`, `Partially`, `Finalized`. Our implementation of the TPC-H benchmark involves a fourth state: `Verified`. Originally, all new orders generated by the TPC-H RefreshFunction1 (i.e., insertion of new orders) are in one of the standard three states, as specified by the TPC-H benchmark. We created a new refresh function, RefreshFunction3, which is executed every time the other refresh functions are executed. RefreshFunction3 changes the status of 99 % of all the non-verified orders to `Verified`.

We use the *completeByNation* sandbox of Listing 3 (Section 3.2) to run queries. We generated up to 25 different sandboxes with different nations. That is, each consumer sandbox focuses on a particular country, specified by the `nationId` parameter. Such a sandbox specifies that the consumer only considers versions of the data lake in which all orders of customers of a particular country are verified. We vary the number of customer sandboxes from 1 to 25, thereby selecting up to 25 different countries to instantiate 25 different customer sandboxes.

5.3 Experimental Environment

All experiments were conducted on a cluster of four machines. Each server possessed two Quadcore Xeon E5-2609 processors and 128GB RAM. The servers were connected with a 10Gbit Ethernet. One server was dedicated to run the Spark coordinator. The remaining three servers were configured to each run eight workers. Each worker had 10GB of main memory. We used Spark Version 1.6.1 and Parquet as a storage format. Parquet is a columnar format with compression and the best format to implement the TPC-H benchmark with Spark. We used Spark's default sort-based shuffle implementation and set all other Spark parameters following best practices.

We carried out two kinds of experiments:

- *Cost:* We measured the cost (in $) using the AWS rates to store the data and to execute the TPC-H refresh functions and check the validity of every sandbox after the execution of these refresh functions. The cost metric is a good metric to measure the *sharing* goal. To reduce dependency on the current AWS rates, we also report on the running times and storage consumption for these experiments.

- *Freshness:* We measure the number of versions of the data lake a query lags behind. That is, when a query is issued by a consumer at Time X and the query is executed using Version Y of the data lake, then the freshness metric is X-Y (lower is better). Both *Personal* and *READY* guarantee freshness so they lag behind 0 versions. However, *Global* can lag behind because the ETL of fresh batches may be postponed because the integrity constraints of a particular customer is not fulfilled. The more consumers, the higher the probability that the ETL process is postponed so that freshness becomes worse with the number of consumers in Global. Freshness is a good metric to measure the *decoupling* goal.

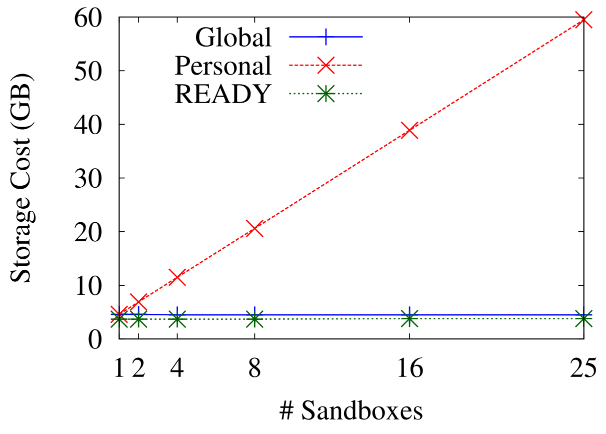

Figure 6: Storage Cost (GB), SF 10

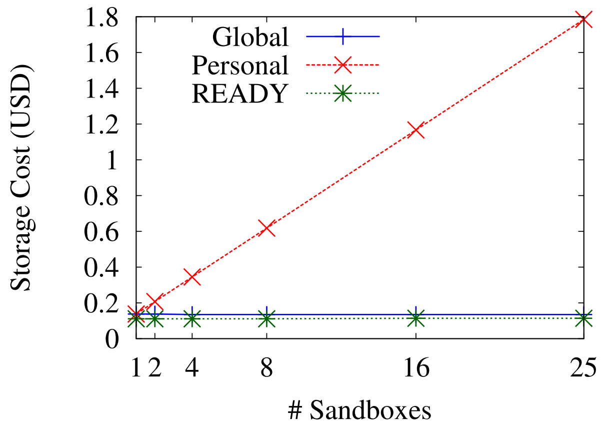

Figure 7: Monthly Storage Cost (USD), SF 10

5.4 Experiment 1: Storage Cost

Figure 6 shows the size of the data lakes in GB for the three different systems (Global, Personal, READY) for a TPC-H database with scaling factor 10 and a varying number of sandboxes. Due to compression in the Parquet format, the size to store a single TPC-H database with scaling factor 10 is significantly lower than 10 GB, the size of the raw TPC-H data. For scaling factor 1, the database sizes are roughly a tenth of the size for a scaling factor 10; we do not show these results for brevity. Figure 7 shows the corresponding monthly storage cost in USD using the AWS rates.

As expected, the storage costs of Personal grow linearly with the number of consumers. The storage costs of both Global and

READY are independent of the number of consumers and sandboxes. Global needs to keep one copy of the database for the producer and one copy for all consumers. READY keeps one versioned copy of the database. READY has higher storage costs than Global because READY needs to keep old versions of data files and can only garbage collect when a data file is not relevant for any sandbox anymore. Overall, however, the differences between the three approaches are small and storage costs were negligible in all our experiments.

5.5 Experiment 2: TPC-H Refresh Functions

Delta Materialization

Figure 8 shows the time in seconds to do Delta Materialization with every execution of the TPC-H refresh functions. The TPC-H refresh functions update about 10 percent of the TPC-H database: For scaling factor 10, for instance, RefreshFunction1 inserts 15,000 new orders and RefreshFunction3 updates about 15,000 orders and sets their status to "Verified". We executed the refresh functions 50 times and Figure 8 shows the average time for each iteration. Accordingly, Figure 9 shows the average cost of Delta Materialization per Refresh Function invocation in USD, using the AWS rates. Overall, the computational costs to execute updates and queries were much more significant than the storage costs.

Not surprisingly, Personal has the highest cost for Delta Materialization and its cost grows linearly with the number of customers because it loads and refreshes a private copy for each sandbox whenever the Refresh Function generates a new version of the data lake that meets the integrity constraint of the sandbox.

READY is much cheaper and its average cost for Delta Materialization is (almost) independent of the number of sandboxes. In any event, READY needs to materialize at most one copy of each version of a data file and this copy can then be shared by an arbitrary number of sandboxes as discussed in Section 4.1. In fact, READY is even cheaper than Global in this experiment because Global maintains two copies of each data file: one for the producer and one for all consumers.

Interestingly, the cost for Delta Materialization drops for Global with the number of sandboxes. This phenomenon is an artifact of our benchmark. As shown in Experiment 4 (Section 5.7), Global does not refresh the consumer data warehouse for more than eight sandboxes because the integrity constraints of all sandboxes are never fulfilled with so many sandboxes. Consequently, Global does not carry out Delta Materialization for the "consumers" with more than eight sandboxes and the cost for Delta Materialization drops with the number of sandboxes.

Incremental Constraint Checking

Figure 10 shows the average cost of checking integrity constraints incrementally (Section 4.2) for the TPC-H Refresh Functions for READY and scaling factor 10. Obviously, this cost grows with the number of sandboxes as more constraints need to be checked. Because of caching effects, however, checking the constraints of 25 sandboxes is only roughly 10 times as expensive as checking the constraints of 10 sandboxes.

Our benchmark sandbox definitions involves a join (Listing 3, Section 3.2) so that checking the constraints was pretty expensive in our benchmark. Inserting a new batch of orders (RefreshFunction1 of the TPC-H benchmark) involved a join of the new orders and the Customer table for every sandbox. To update the counters for the other refresh functions, we kept the *ids* of all orders that has a status other than "Verified" for each sandbox and READY joined this set of *ids* with the *ids* of the updated orders. Of course, READY can

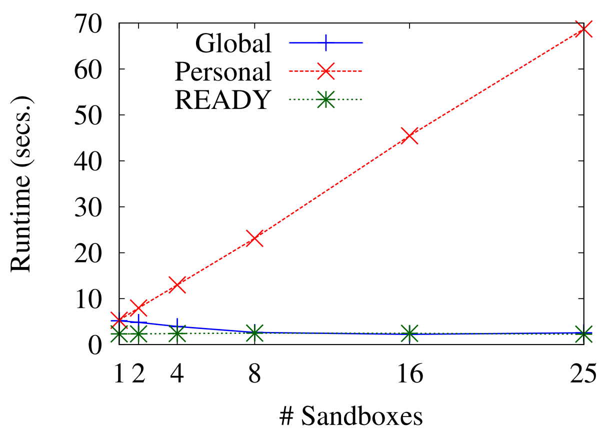

Figure 8: Refresh Functions (secs), Delta Mat., SF 10

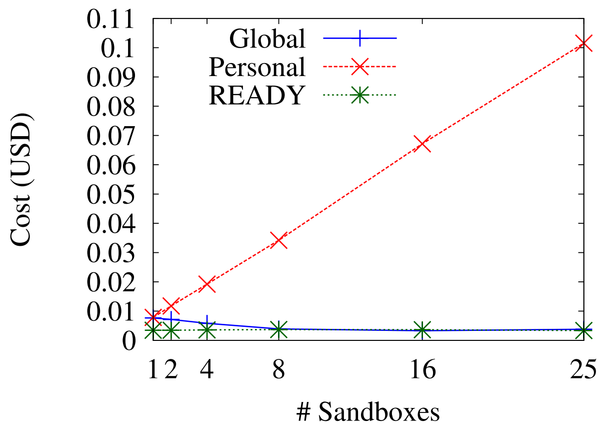

Figure 9: Refresh Functions (USD), Delta Mat., SF 10

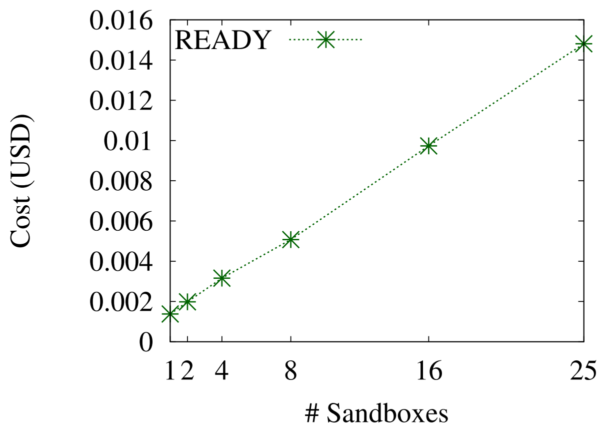

Figure 10: Refresh Functions (USD), Checking, SF 10

check simpler integrity constraints much faster and cheaper.

Comparing Figures 10 and 9, it becomes clear that the cost of incrementally checking integrity constraints is negligible compared to the cost of Delta Materialization. That is why we spent considerable more time to optimize Delta Materialization and to make it scalable in our implementation of the READY prototype. However, checking integrity constraints can be expensive, too, if not done right. Lazy constraint checking, for instance, is expensive, roughly in the same order as *initialization* which is studied in the next subsection.

The cost to do incremental constraint checking is almost identical for Personal and Global as for READY; so, we do not show these results for brevity. In all of these approaches, the main effort of incremental constraint checking is to maintain the materialized views for all sandboxes (i.e., the counters) and this cost is independent of the approach or whether the integrity constraints are met or not. For lazy constraint checking (not shown), Global is cheaper than Personal and Ready because Global benefits from short circuiting: Global checks the conjunction of the `WHEN` clauses of *all* sandboxes and it can stop checking as soon as one of the predicates of one sandbox fails. That is, if Sandbox 1 is not compliant, Global does not need to check the constraints of any other sandbox. In contrast, Personal and READY must check the `WHEN` clauses of all sandboxes separately.

Initialization

When a new sandbox is created the counters (and *id* sets) need to be initialized for incremental constraint checking. In READY, this initialization requires a scan through the whole database to evaluate the complete `WHEN` clause. For our benchmark sandboxes (Listing 3), this initialization took about 14 secs for a TPC-H database with scaling factor 1 and about 23 secs for a TPC-H database with scaling factor 10. So, creating a new sandbox and initializing it for incremental constraint checking is an expensive operation in READY.

Like constraint checking, this initialization cost to set up the counters is the same for READY, Personal, and Global: It is fundamental to any system that supports multiple sets of integrity constraints in a data lake. For Personal, there are substantial additional costs to materialize and create the “Personal” copy of the new sandbox which are in the order of the size of the database. So, again, Personal is by far the most expensive approach for accommodating a new class of customers with a specific set of integrity constraints.

Scalability

Figure 11 shows how the cost of executing the refresh functions varies with the scaling factor for each approach. The figure shows the aggregate of the cost for Delta Materialization (Figure 9) and Constraint Checking (Figure 10). While the cost of checking integrity constraints grows with the size of the database (in our benchmark a larger join needs to be executed), we expect the cost to be fairly flat and independent of the database size because overall the costs are dominated by Delta Materialization. The cost for Delta Materialization should be independent of the size of the database because READY only creates a new version of the data files that contain updated records and the size of each data file is constant and independent of the total size of the database. In our experiments, for instance, we had 117 data files of size 2 MB to store a TPC-H database with scaling factor 1 and 1170 data files of size 2 MB to store a TPC-H database with scaling factor 10.

As shown in Figure 11, the average cost to execute the TPC-H Refresh Functions is actually smaller with scaling factor 10 than with scaling factor 1. The reason is again an artifact of our benchmark and our benchmark sandbox definitions. As shown in Experiment 4 (Section 5.7), the likelihood that a version of the data lake satisfies the integrity constraints of a sandbox decreases with the size of the database in our experiments so that we need to carry out Delta Materialization less often to execute the TPC-H Refresh Functions in a database with scaling factor 10 than in a database with scaling factor 1.

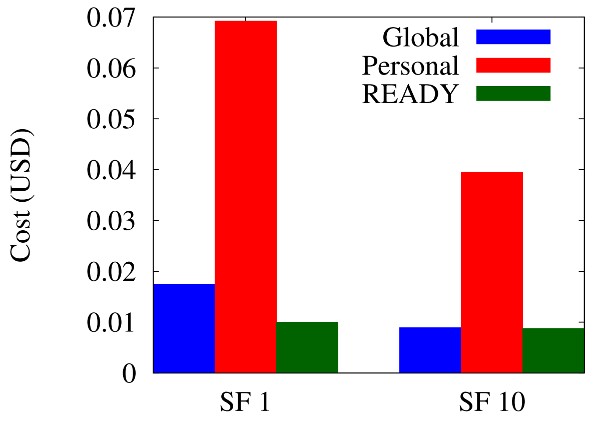

Figure 11: Refresh Functions (USD), Vary SF, 8 Sandboxes

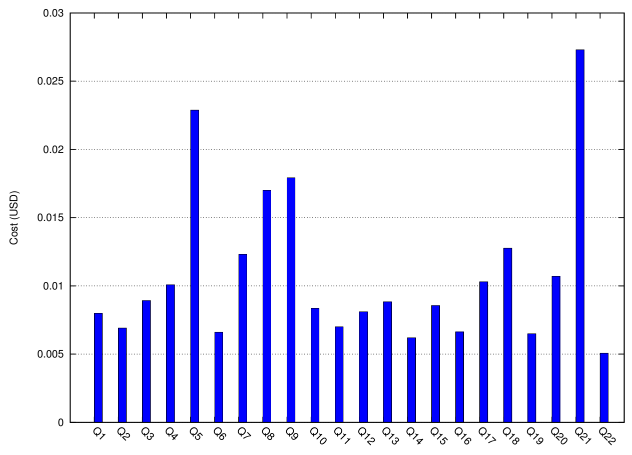

Figure 12: READY: Queries (USD), SF10

5.6 Experiment 3: TPC-H Queries

Figure 12 shows the average running times of executing all the TPC-H queries with READY and a scaling factor of 10. These numbers are not READY-specific and are the same for Personal and Global or just regular SparkSQL without any integrity constraints. Due to the coarse-grained approach to implement versioning, READY has no overheads when it comes to processing queries. All that READY needs to do is to use the *VersionMap* to find all data files that are relevant for a specific sandbox and execute the query on these data files.

Comparing Figures 12 and 10 relates the overheads of READY (constraint checking) to the cost of executing queries. It becomes clear that READY is affordable and that the overheads of adding integrity constraints to a data lake are roughly in the same order or even less as answering a few queries in the data lake (depending on the scale of the database and complexity of the query). With a

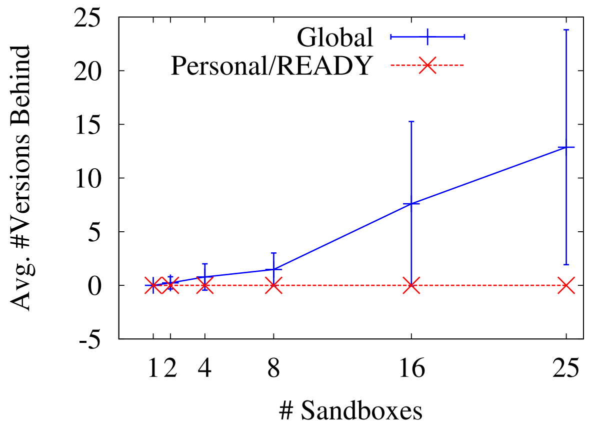

Figure 13: Data Freshness, SF1

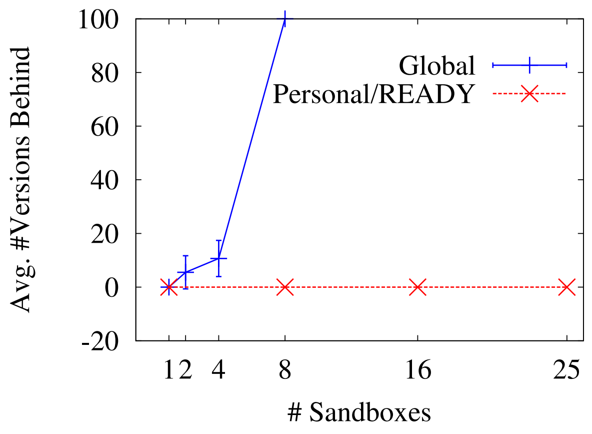

Figure 14: Data Freshness, SF 10

growing query workload, the overheads of READY become negligible compared to the total cost of ownership which is dominated by the cost to execute queries.

5.7 Experiment 4: Data Freshness

Figure 13 shows the average number of versions a query lags behind for a TPC-H database with scaling factor 1. READY and Personal guarantee freshness as the integrity constraints of one consumer do not block any other consumer. In contrast, Global does poorly with regard to freshness with a growing number of sandboxes as all integrity constraints of all consumers must be visible before a batch of updates with new orders becomes visible. Furthermore, the variance is high because we might get lucky and the updates become visible immediately or unlucky.

Figure 14 shows the data freshness for a TPC-H database with scaling factor 10. This experiment shows that with Global and a large number of sandboxes, it might even happen that updates never become visible. In this experiment, there is always some order that has not yet been verified and relevant to some sandbox so that with more than eight sandboxes no version of the data lake meets the integrity constraints of *all* sandboxes. That is why ETL processes of data warehouses are crafted very carefully in practice.

6. RELATED WORK

This work builds on top of a great deal of related work. First, the READY prototype makes use of temporal database technology to execute a query on the right version of a data lake [15]. Thus, all temporal query processing techniques are relevant; e.g, [8]. Second, all techniques to incrementally maintain views are important to achieve good performance with READY; e.g., [3, 1].

Another related area of research is work on data completeness and data quality [11, 13, 9]. Motro et al. studied how to define query entailment and verification by verifying the database metadata [11]. More recent work discussed the impact of missing data records on query results [13, 9] and how it impacts the monetary costs [9]. Libkin et al. [10] show that the current SQL semantics are prone to deliver incomplete results when null values are present in the data set. That work proposes improvements to the SQL standard and SQL processors to achieve completeness. All this work, however, focused on *data completeness* such as missing records and missing values. In contrast, our work considers an orthogonal aspect: *semantic completeness* defined by integrity constraints.

In the recent years, there has been a great deal of interest on data quality issues in data lakes. For example, Constance [7] helps improving data integration tasks within data lakes by providing users a unified interface for query processing and data exploration. Another example is the CLAMS system [4]. CLAMS focuses on data quality issues with unstructured and semi-structured data, where there are no integrity constraints. CLAMS automatically discovers data quality rules over such semi-structured data. Again, all that work is orthogonal to our work on READY.

Arguably, the most related system is DataHub [2]. Even though DataHub was developed for a totally different purpose, with different scenarios in mind, and, thus, different technical contributions, DataHub and READY share some interesting features such as complex versioning and sharing of data.

7. CONCLUSION

This paper studied how to add integrity constraints to data lakes. The most important observation is that it is not enough to simply support one set of integrity constraints (as done by relational database systems today). Instead, the sharing nature of data lakes gives rise to supporting multiple sets of integrity constraints and expose each set in a separate sandbox. This paper studied the basic principles of sandboxes, a first scenario that applies these principles (producer / consumer information flows for data warehousing), and the results of preliminary experiments that study cost and data freshness trade-offs.

There are a number of avenues for future work. One area is to study more complex models such as multiple producer sandboxes and branching (Section 3.4). Another important area is performance; there are many ways to improve the performance of the basic algorithms presented in Section 4; in particular, for special classes of integrity constraints such as completeness constraints with universal quantification. One particular area of optimization that READY could benefit from is "multi-query optimization" because there are typically many common sub-expressions across the integrity constraints of different sandboxes that READY could exploit. READY would already improve greatly by implementing a shared scan to evaluate integrity constraints.

8. REFERENCES

[1] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

[2] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[3] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In C. Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986.*, pages 61–71. ACM Press, 1986.

[4] M. H. Farid, A. Roatis, I. F. Ilyas, H. Hoffmann, and X. Chu. CLAMS: bringing quality to data lakes. In Özcan et al. [12], pages 2089–2092.

[5] A. Gärtner, A. Kemper, D. Kossmann, and B. Zeller. Efficient bulk deletes in relational databases. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 183–192, 2001.

[6] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. pages 157–166, 1993.

[7] R. Hai, S. Geisler, and C. Quix. Constance: An intelligent data lake system. In Özcan et al. [12], pages 2097–2100.

[8] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1173–1184. ACM, 2013.

[9] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1275–1286. ACM, 2014.

[10] L. Libkin. Incomplete data: what went wrong, and how to fix it. In R. Hull and M. Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 1–13. ACM, 2014.

[11] A. Motro. Integrity = validity + completeness. *ACM Trans. Database Syst.*, 14(4):480–502, 1989.

[12] F. Özcan, G. Koutrika, and S. Madden, editors. *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 2016.

[13] S. Razniewski, F. Korn, W. Nutt, and D. Srivastava. Identifying the extent of completeness of query answers over partially complete databases. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 561–576. ACM, 2015.

[14] R. T. Snodgrass. The temporal query language tquel. *ACM Trans. Database Syst.*, 12(2):247–298, 1987.

[15] R. T. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, 1986.

[16] S. Subramanian, S. Bellamkonda, H. Li, V. Liang, L. Sheng, W. Smith, J. Terry, T. Yu, and A. Witkowski. Continuous queries in oracle. In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas, and E. J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1173–1184. ACM, 2007.