Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 115

Systems Group, Department of Computer Science, ETH Zurich

Completeness is in the eye of the beholder
A sandbox concept for databases

by

Moritz Hoffmann

Supervised by

Prof. Donald Kossmann

March 2014 – September 2014

inf | Informatik
Computer Science

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Completeness is in the eye of the beholder
A sandbox concept for databases

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Hoffmann	Moritz

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 3. September 2014	Moritz Hoffmann

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

Recently there is a strong tendency in companies to combine databases into a single data repository. In this work, we propose a data completeness model which provides a means for applications to define constraints that specify when data relevant to them is complete. We introduce the sandbox concept which provides the impression of a local database to applications. In addition, we propose an implementation, introducing different optimization approaches for completeness checking. Finally, we show the viability of the sandbox concept for ensuring data completeness.

Contents

List of Figures

1 Introduction

Today, nearly all businesses use databases to store the data required for operations and analysis. Traditionally, data is stored in relational databases separated by department and functionality. For example, Finances has its own databases, customers are managed using a Customer Relationship Management (CRM) system and Procurement maintains items in storage. In addition to that, a company's performance is measured periodically and yearly statements have to be published. For each of these tasks a separate set of databases exists.

Between databases there often exists some degree of redundancy. Customers may appear in several databases, as well as orders and items in orders. A complete view of a company's data is available only when combining all databases. Redundancy can cause increased maintenance costs and might lead to data corruption if not enough care is taken. Ensuring consistency is also more complicated if data is spread over several databases.

Sharing a common subset of data is hardly, if at all, possible, because this might conflict with the various responsibilities in companies.

Combining databases to overcome these problems also proves to be a difficult task. Next to business related reasons, e.g. each department wanting to have its own data under its control, it is also problematic from a technical point of view. Usually, companies have a long-grown IT infrastructure, with proven and established well-functioning applications. Creating a unified database would require all these applications to be moved to the new database. Such a migration could provoke lots of possible disruptions.

Apart from the work involved to perform the actual migration, there also might arise compatibility problems, meaning new application software versions might need to be acquired, or the application software might need to be replaced altogether. Moreover, the applications might have incompatible or diverging data consistency requirements, which are hard to deal with if the unified database cannot fulfill all of them. When we also want to ensure continuous service, this adds another difficulty.

A typical example for completeness is business performance analysis, which is performed at fixed time intervals, say every month. It is assumed that at a given point in time all required data is in a complete state. Here, a problem arises: If this assumption is too optimistic, we might query the data too early and have to rerun the same analysis, or worse, the analysis is wrong. If the assumption is too conservative, we might delay the analysis for no reason.

Imagine a manager wanting to know how well a branch of a multinational company performed last month. The company's upper management performs an overall analysis each month after all countries have closed all transactions. While some country's branches work faster and their results are available almost immediately, others are slower. The company-wide analysis can only be performed when all countries have finished their transactions. Although the manager is only interested in a specific branch, all branches have to report their data first. Only then, the specific branch's data is available. So, the global completeness constraint delays access to a specific subset of the data. To overcome this limitation, the manager could define personal completeness constraints and perform the analysis for a specific branch as soon as the data is ready, possibly saving a lot of time and in turn being able to run the business more efficiently. By only exposing the data belonging to a particular branch, which is in itself complete, we are sure that the analysis will be limited to only complete data and can be assumed to be correct.

In contrast to this we are now looking at firefighters. Assume there exists a database that contains information about buildings, addresses and other details in a city, as well as information about current incidents. From the firefighters' perspective, every information might be relevant – they are interested in all data, even if it might be incorrect or incomplete for someone else! Firefighters should be able to access all information and left to decide themselves what to do with it. They might also be interested in the *absence* of information, which is only possible to recognize if all known information is present.

Another use case is related to software engineering. In a typical software engineering process programmers develop parts of software which belong to some milestone or planned release. In this use case we focus on issues (bugs) blocking the release of a product. The software can only be released once the milestone is reached, and that in turn is the case when all associated bugs are closed (resolved). In other words, a product can go to the next development phase once bugs for the previous phase are closed. It is a manager's decision to advance to the next phase. In traditional data management systems, this is done by checking if all bugs for the current milestone are closed, and when this is true, manually advancing the development efforts to the next milestone.

Using our proposed system, the fact that all bugs must be closed for a specific milestone would be expressed as a completeness requirement. As soon as the bug database fulfills this requirement, it would notify the manager.

All use cases share a common problem: The notion of *completeness* is subjective to each user and highly depends on individual needs and expectations. Furthermore, each user might be interested to see different parts of the data. Thus, the *visibility* of the data also needs to be based on subjective needs. The use cases and introduction showed that both the time when something is visible as well as the content of what is visible matter. The focus of this work is to analyze how to design and build a system that allows users to express completeness constraints and provides them with the data matching these constraints.

1.1 Sandbox

A possible solution for the problems described above lies in the introduction of a software sandbox between a database and a user.

A sandbox refers to a box full of sand that children play in. They can do whatever they want because all sand (in theory) stays inside the sandbox and the area around it is not affected. In that sense, a sandbox is a little isolated world of its own and the player interacts only with the environment provided by the sandbox.

In computer science, a sandbox is a well known concept, especially in security. It is often used in operating systems to protect other parts from the effects of an application. This can be used to execute untrusted code because the sandbox guarantees that nothing gets out.

Similarly, we apply the sandbox concept to relational databases. In this case, however, we want to isolate the user inside the sandbox from the outside, protecting them from data they do not expect and cannot understand: The users only get to see what they want to see. In this sense, the sandbox concept used here focuses more on usability than on security.

A sandbox acts like a traditional database by restricting *what* is visible and showing it *when* it is complete. Both aspects are based on a user's subjective expectations. A sandbox is potentially backed by a heterogeneous database system. Only when the data reaches a *complete* state, it will be visible inside the sandbox. By providing a traditional database interface, we are able to solve a lot of the integration problems.

Applying the sandbox concept to databases has other benefits. For example, in a company might exist several databases for specific tasks. We merge the data into one big shared database, at the same time providing a sandbox for each application to give it the illusion of having a regular database. The task of the sandbox is to define *what* data to make visible, i.e. make sure that the data is matching the application's consistency requirements, and *when* to provide the data. For a user accessing data through a sandbox it does not matter whether behind the sandbox is a unified database or a plain old Structured Query Language (SQL) database because the sandbox provides an abstraction to it.

To sum it up, a sandbox behaves like a traditional well-known database towards applications and is an instrument allowing the user to formulate data completeness constraints. It limits the exposure between underlying data and the user. The underlying database system, which can range from a single database to a multi-tier heterogeneous architecture, is irrelevant to the user of a sandbox.

2 Related work

To our knowledge, there are no similar data completeness approaches comparable to what we present in this work. The main reason for this is the fact that we propose to let the user specify completeness constraints to be checked by the database on the user's data. In addition to that, we provide means to let the user work with his data at versions he considers to be complete. Nevertheless, our research touches other fields that have partial similarities and aid us in solving and differentiating our work. In the following, we will describe some of those fields.

2.1 Data completeness model

In data warehouse theory, data completeness is supposed to make sure that all necessary data has been validated and loaded into the target repository. In order to accomplish this, the database might have to compare record counts between source and target, check for rejected records, verify the uniqueness of values and also check that values remain within their boundaries. However, these requirements might not be met at all times because errors occur and as a result, users might be blocked, or worse, work with incomplete data. In the last few years, there have been research initiatives to model and define how data completeness should be handled. An interesting field of research is to logically determine if a query to the database can be answered completely. Different authors [12, 14–17] argue that queries could be logically verified to determine if the database being queried could provide a complete answer to a query. Savkovic et. al [19] created a system capable to verify query entailment using database's meta data. Their system creates a disjunctive logic program from a query and database meta data to determine if a query's result is complete. If the program can determine completeness, then a query's response can be considered complete. This is a very interesting approach for data warehouse based applications as it can be used to automate completeness checking within the database.

Our work is based on the idea that completeness is subjective, i.e. each user has a different notion of completeness which cannot be determined by software just based on the data.

Libkin [13] focuses on determining how to deal with incomplete data in the presence of null values. In his work, he analyzes how null values affect queries posed over databases assumed to be complete, but also proposes how this problem could be handled by enhancing query processing techniques. His work describes new complementary research

paths that need to be solved for data completeness problems in presence of null values. Compared to this, our work proposes to create a completeness model solely based on user's needs and constraints.

2.2 Completeness checking

There exist many ways to evaluate conditions based on the data in a database. We consider the following approaches relevant to specific conditions that could also cover some completeness constraints. However, we do not see that any of those approaches completely covers our approach.

Incremental computation Incremental view maintenance has been around for many years and is a well studied subject. It is an optimization to keep materialized views up to date by incorporating changes from transactions. The idea is to update query results only by applying the deltas of relations to the query and that way computing a delta of the materialized view. Deltas are supposed to be small and because of this, a huge performance improvement might be realized [21]. We are using parts of this approach for completeness checking, but optimize it even further as we are only interested in the cardinality of a relation, which we compute as discussed in 3.8 on page 33.

Stream processing Stream processing, also known as complex event processing (CEP), is a technique to evaluate the result of a formula over a stream of data using a sliding window. When checking a formula over an infinitely small window this is semantically equivalent to our proposed completeness checking approach. However, there are situations where stream processing is not suitable, especially when evaluating joins that depend on data not being changed in the window but preexisting in the database. Also, it is in general not aware of versions and transactions, so it might tell the user that *some* version was complete, but not which one. Using a combined incremental and streaming approach could be an interesting field to investigate. An overview of this approach is given by Cugola and Margara [5].

Temporal data management We use a simple temporal table model to access past data without special indexes and optimizations besides regular database indices. Temporal data management researches how to efficiently store and access temporal data using specialized storage and index structures. It is a field with very active research. Snodgrass' temporal data model [20] started new research [6, 18]. New index structures have been proposed by [2,4,6,10,11,22]. Temporal data management could be useful to reduce the costs of past processing, i.e. checking data for completeness that is in the past.

Active databases An active database is a relational database with additional features to define events and responses. It is a different concept to our completeness model as

it defines a very limited set of events and the user has to be aware of what events could make the desired database version complete. The work presented by Behrend et al. [3] describes an SQL extension to active database research for performing time-based triggers. Their work tries to complement [8] by allowing users to trigger actions by time events, to repeat the same action during a time range, and delaying the reaction execution if needed. This through SQL. In comparison, our model does not require users to define events at which the completeness constrains are to be evaluated. Those events could be derived from the completeness constraints themselves.

3 Approach

In this chapter, we first present a theoretical foundation of the sandbox concept and an SQL syntax extension to work with sandboxes. Next, we show how our use cases can be expressed using the proposed notation, followed by sections covering implementation approaches and optimization. At the end, we present our benchmarking results.

3.1 What is completeness?

Completeness is a state of the data that matches a user's expectation in order to be able to work with the data. As it covers a user's expectation, it is personal in nature and does not reflect a ground truth. In this work, completeness is defined over the contents of a database, and a database can either be in a complete or incomplete state according to the user's expectations.

Completeness is an orthogonal concept to SQL integrity constraints as it does not impose restrictions on what data the database is allowed to contain. SQL considers a database to be complete when integrity constraints are met. Therefore, integrity constraints are not suitable to express completeness conditions, because this would require the data to be complete at all times. User defined completeness conditions can be seen as a generalization of integrity constraints. Completeness constraints may be defined over a subset of a given database, and this may also be what a user is interested in.

The following section about the theoretic foundation expresses this formally.

3.2 Theory

We define the sandbox concept on top of relational databases. We extensively make use of relational algebra, which defines relations, tuples, and operations to derive new relations from existing ones. The following description defines concepts related to sandboxes.

3.2.1 Informal definitions

The following items list concepts that are related to sandboxes, only defining them informally. In the 3.2.2 we provide mathematical definitions of the concepts explained here.

Database A database is a set of distinct relations. We use DB to denote a database.

Relation schema A relation is characterized by its relation schema, which defines the attributes of tuples within that relation.

Relation A relation is a set of tuples which all conform to one relation schema.

Version Since databases change over time, we identify different database versions using version numbers. The version number is monotonically increasing and can be understood as a logical time stamp. For the database DB at version number i we write DB_i.

Tuple instance The basic unit in the database, which is an ordered list of elements conforming to a relation schema. It belongs to a specific relation and can appear in different database versions.

Sandbox A sandbox is defined by a set of functions on relations and a global completeness predicate. The relation functions determine *what* is visible, and the completeness predicate determines at what points in time it is visible (*when*).

We are using existential ($\exists$) and universal ($\forall$) quantification to denote that there is at least one matching element in a set or that a predicate holds for all elements in a set. To build new sets, we use the set-builder notation, for example to construct the set Y from a set X with a predicate p we write: $Y = \{x : x \in X \land p(x)\}$.

3.2.2 Database

Versions number A version number represents a logical time stamp (monotonically increasing). The set V denotes all version numbers that are present in the database.

Relation A relation r is defined by a set of tuples that all follow the same relation schema. Furthermore, a relation r that is contained in database version DB_i is denoted by r_i.

Database version A database DB_i is a set of relations at version number i. The set DB_V contains all DB_i where i is a version number:

$$DB_V = \{DB_i : i \in V\} \tag{3.1}$$

We assume that there are no schema changes on relations between database versions.

Transaction A transaction describes how to derive a new database version from an existing one. We define the effects of a transaction on the relation level. Given a relation r, the version at the start of a transaction is $\mathsf{pre}(r)$ (*pre*), the version after is $\mathsf{pst}(r)$ (*post*). The set of changes (removed and inserted tuples) is called $\mathsf{dlt}(R)$ (*delta*), see 3.8.1 for a more elaborate definition, where we also define the $\uplus$ operator. It holds that

$$\mathsf{pre}(r_i) = r_i \tag{3.2}$$
$$\mathsf{pst}(r_i) = r_{i+1} \tag{3.3}$$
$$r_{i+1} = r_i \uplus \mathsf{dlt}(r_i) \tag{3.4}$$
$$\tag{3.5}$$

It follows for relation r at any version that

$$\mathsf{pst}(r) = \mathsf{pre}(r) \uplus \mathsf{dlt}(r) \tag{3.6}$$

3.2.3 Sandbox

Sandbox A sandbox SB is a pair $\langle P_r, p_s \rangle$ of a set of relation functions P_r and a completeness predicate p_s. The relation functions P_r specify which tuples will be provided to the user at a given database version, if the version fulfills the completeness constraint p_s.

$$SB = \langle P_r, p_s \rangle \tag{3.7}$$

Relation function A relation function is a function that maps a relation r_i to a subset of that relation r'_i. A sandbox's relation functions are contained in P_r. Then,

$$\forall p \in P_r : r_i \xrightarrow{p} r'_i. \tag{3.8}$$

Completeness predicate A completeness predicate is a function to determine if a database at version i is complete. It returns a boolean value and is contained in p_s:

$$DB_i \xrightarrow{p_s} \{\top, \bot\}. \tag{3.9}$$

Sandbox completeness A sandbox SB is complete in version i if its completeness predicate $p_s(DB_i)$ holds.

Sandbox contents Applying a sandbox $SB = \langle P_r, p_s \rangle$ to a database DB_i, the sandboxes' contents SB_i is defined as

$$SB_i = \{\rho(r_i) : r_i \in DB_i \wedge \rho \in P_r\}. \tag{3.10}$$

Note that the sandbox contents are independent of the completeness condition.

Completeness predicates

A completeness predicate is a boolean function taking a set of relations as input. It is used to decide whether a particular database version is complete or not. In general, there are two forms of completeness predicates (or any combination of those). Existential quantification is used to check that there exists a certain tuple in the database, and universal quantification is used to verify that a set of tuples conforms to some specification. In this analysis we focus on completeness predicates having one of the following forms, where F is some relation, t a tuple of relation S and p is a predicate.

$$\exists t \in S \tag{3.11}$$
$$\neg\exists t \in S \tag{3.12}$$
$$\forall t \in S : p(t) \tag{3.13}$$

We also consider completeness predicates that require a certain number of tuples to be present or absent. σ is the selection operator as defined in relational algebra. Then,

$$\exists t \in S \Leftrightarrow |S| \geq 1 \tag{3.14}$$
$$\neg\exists t \in S \Leftrightarrow |S| = 0 \tag{3.15}$$
$$\forall t \in S : p(t) \Leftrightarrow |S| - |\sigma_p(S)| = 0 \tag{3.16}$$

The correctness of those equalities follows directly from the definitions of universal and existential quantification. Universal quantification says that a predicate has to hold for all elements in a set in order to be true, existential quantification requires that at least one element exists where a predicate is true. Hence, we can test the cardinality of a set containing tuples matching a predicate to be greater or equal one in order to determine if the existential quantification is true. A similar argument holds for universal quantification: We can compare the count of all elements in a set to the count of elements matching a predicate taken from the same set. If the counts are equal, we know that the predicate holds for all elements in the set.

The type of existential completeness constraints we analyze in the following have the following form, where $op \in \{<, >, =, \ldots\}$ and $n \in \mathbb{N}$:

$$|S| \; op \; n \tag{3.17}$$

3.2.4 Completeness

Data completeness defined as a predicate over a database version. Figure 3.1 shows two completeness predicates p_1 and p_2 evaluated on different database versions.

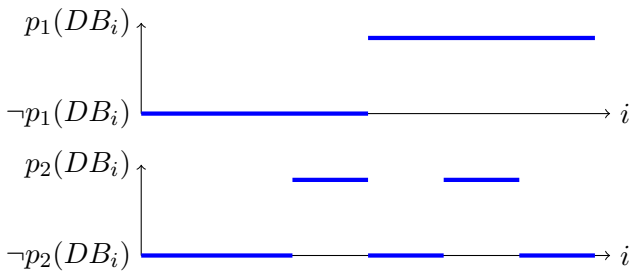

Figure 3.1: Two examples of completeness functions for different database versions i. The predicates are independent of each other. Note that based on the completeness function, a database version can be complete at one point in time and become not complete again at a later point in time.

3.2.5 Sandbox goals

The sandboxes are defined in the theory section (3.2). When using our sandbox concept combined with relational databases, we have the following goals:

Low cost Like all operations executed on a database, creating and using sandboxes is associated with a cost. This cost depends on a variety of factors, such as the amount of data in the database, the transaction rate, the complexity of the completeness predicate and the number of sandbox instances in the system. In general, the cost needs to be as low as possible. We would like the cost to follow a predictable pattern to be able determine where such costs will arise – if the software allows that.

Unaltered behavior For the regular operation of a database, it is crucial that its behavior remains unaltered, meaning an update must have exactly the same effect as it does on a database without sandboxes. This means that sandboxes are allowed to slow down transaction processing as explained above, but the transaction behavior must not be altered. No new rollbacks may be introduced by adding the sandbox concept to the database.

3.3 SQL extensions

SQL is the de-facto standard to interact with databases. It defines a syntax to query, update and delete data as well as defining schemata. We are using SQL as a tool to allow users to interact with sandboxes.

Listing 3.1: Creating a sandbox

```
1 CREATE SANDBOX sandboxName [ ( IN argname argtype [, ...] ) ]
2 [ WHEN
3     completenessPredicate ]
4 [ WITH
5     relationName: predicate [, ...] ];
```

3.3.1 Creating a sandbox

From an SQL perspective, a sandbox consists of two components: the sandbox definition, and, associated to it, a sandbox instance. The sandbox definition describes the properties of a sandbox while the instance applies those properties on the database.

The syntax to create a sandbox is aligned with standard SQL syntax. Listing 3.1 shows the syntax to create a new sandbox.

In the following we describe optional configuration options for the sandbox definition, which can be provided in addition to the required unique name that identifies the sandbox.

Completeness constraints are specified as a WHEN condition. They are expressed as a boolean formula that is true once the database reaches a complete version. If no completeness constraint is provided the sandbox is considered to be complete at any version. This expresses the p_s of a sandbox definition.

Visibility predicates select data that is visible from within a sandbox, expressed as predicates applied on relations. They are expressed after the WITH keyword. If no visibility predicate is supplied for a given relation the sandbox will return all tuples within that relation. This expresses the set of relation functions P_r of a sandbox definition.

Parameters can be declared for a sandbox and can later be bound to values when using the sandbox. This concept adds some flexibility to sandboxes and is similar to function parameters. It is illustrated in the business intelligence example explained in section 3.4.1 on page 19.

3.3.2 Instantiating a sandbox

Listing 3.2 shows how to execute the SQL statements referred to as *user-provided statements* within a transaction. Note that these statements are only executed if there is

Listing 3.2: Starting a transaction based on a sandbox

```
1 BEGIN ... USING SANDBOX
2     sandbox_name [ ( value [, ...] ) ]
3     { NOW | { NEXT | LAST } [ BETWEEN start AND end ] };
                                        ▷ Temporal completeness operator
4 ...;                                  ▷ User-provided statements
5 { COMMIT | ROLLBACK };
```

a database version that is complete according to the sandbox completeness constraint.[1] Parameters need to be provided if the sandbox defines them.

Temporal completeness operator

The temporal completeness operator describes how to test database versions within the specified time range. Users can choose between NOW, NEXT and LAST. The NOW-operator instructs the database to test whether the current version of the database is complete, and only if this is the case, the transaction is started, or the command executed. Otherwise, an error is raised. The NEXT temporal completeness operator searches for the first complete state in the future by waiting for the first complete version to appear, starting at the current version.[2] The LAST operator performs the opposite: It searches backwards in time, also starting from the current version.

Both NEXT and LAST allow for an interval to be specified. If provided, the search is only performed within that interval.

Whenever we use time spans/intervals their start is inclusive but the end is exclusive. For example, a time span from version u to v includes u but not v, e.g. $[u, v)$.

Continuous completeness queries

An orthogonal alternative to starting a transaction when completeness is reached is to let the database notify a user via a different channel that completeness has been reached. The transaction approach limits the number of complete versions that can be observed to a maximum of one.

If, on the other hand, an application wants to be notified of more than one complete version without explicitly starting another transaction, the continuous mode of operation would be the right choice. An application subscribed to a continuous sandbox would be notified whenever a complete version is reached. A version number can be used to

[1] The temporal completeness operator determines which database versions are checked for completeness.
[2] If no interval is specified, the search for a complete state will only stop when the database becomes complete.

Listing 3.3: For-all syntax proposal

```
1 FORALL (SELECT ... FROM ... WHERE ...)
2     SATISFY ...
```

identify a particular version to enable AS-OF style queries later on. In this case the database does not need to maintain the visibility constraints for individual relations. As this concept does not fit within the command-response concept of SQL, we do not specify a syntax for it.

3.3.3 For-all operator

We have seen the need to express universal quantification conditions when stating completeness conditions because there are cases in which a certain condition is met only if every member of the domain satisfies it. SQL itself does not have an explicit for-all operator, it only supports existential quantification. For convenience, we introduce a for-all operator based on the following mathematical definition. Listing 3.3 also presents a proposed syntax extension. For-all is the same as there does not exist an element for which the predicate does not hold, as shown in the following formula.

$$\forall x \in X : p \Leftrightarrow \neg \exists x \in X : \neg p \tag{3.18}$$

3.4 Use cases

With the sandbox concept, we can implement use cases where actions of a program depend on the states of data in a database. If the data matches a condition, an event is triggered allowing a client to run a single statement or a transaction on that particular complete version. Depending on the database capabilities, the database can also remember the complete version and allow to query it at a later point in time. We show how to express completeness constraints and requirements of the use cases using the SQL syntax we propose.

3.4.1 Business intelligence

As described in the introduction (cf. p. 6), we want to model a simple business with orders, items and customers. We are interested in the company's performance at the end of a month, which can only be calculated when all transactions are closed. Waiting for the first complete version is achieved by using the NEXT temporal completeness operator.

Listing 3.4: Business intelligence use case, create sandbox

```
1 CREATE SANDBOX BusinessIntelligenceSandbox(
2     IN :startDate date, IN :endDate date)
3 WHEN
4     FORALL (SELECT status FROM Order WHERE date BETWEEN
5         :startDate AND :endDate)
6     SATISFY status = "processed"
7 WITH
8     Order: data BETWEEN :startDate AND :endDate;
```

Listing 3.5: Business intelligence use case, use sandbox

```
1 BEGIN USING BusinessIntelligenceSandbox("2014-08-01", "2014-08-31") NEXT
2     BETWEEN "2014-09-01" AND "2015-01-01";
3 ...
4 COMMIT;
```

We limit our scope by looking at a single *Order* relation which encodes individual orders by customers and stores information about whether the order is completely processed or not. We are not trying to model a whole system, as this is not required to show the sandbox functionality.

An order has an identifier, a customer, a date and a status field. The status can either be new, processing or processed:

$$Order(id, customer, date, status)$$

The sandbox is defined as being complete if all orders over a time span have been processed. This time span can be used as a parameter (See listing 3.4 for the create-sandbox instruction). The sandbox also only exposes complete data matching the visibility predicates, which are orders having their processing date in the specified time span. It is worth noting that a real sandbox would also restrict other relations to keep foreign key relationships valid. The time span's end is not checked for completeness, according to the specification of time spans in this work (cf. p. 18).

To actually run the business intelligence queries, the user would instantiate the sandbox. This involves binding the parameters of the sandbox to concrete values. In listing 3.5, we show how to start a transaction as soon as the sandbox is complete, i.e. when the next complete version has been found.

Listing 3.6: Bug tracker sandbox

```
1 CREATE SANDBOX CompleteMilestone(IN :milestone int)
2 WHEN FORALL (SELECT status FROM Bugs WHERE Bugs.milestone =
      :milestone)
3     SATISFY status = "closed"
4 WITH
5     Bugs b: b.milestone = :milestone;
```

Listing 3.7: Bug tracker sandbox instance

```
1 BEGIN USING CompleteMilestone(15) NEXT;
2 ...;
3 COMMIT;
```

3.4.2 Bug tracker and universal quantification

In this scenario, we model the issue tracking part of a software engineering project. We assume the existence of software issues (aka bugs), which are associated to a milestone table. Each bug is also associated to a single component. When a bug is created, it is in state open. After solving it, it is closed. The *Bugs* relation has the following form:

$$Bugs(b\_id, component, milestone, status, \cdots)$$

A *Milestone* has an identifier. Other fields are not relevant to this use case.

$$Milestone(m\_id, \cdots)$$

A milestone is complete as soon as all bugs associated with it are closed. For a given milestone M, this can be expressed as

$$\forall b \in \sigma_{milestone=M}(Bugs) : b.status = \text{"closed"}$$

Listing 3.6 shows the sandbox expressing this constraint.

We can imagine at least two ways to use the sandbox: First, to be notified when a release can be created, this would mean to wait for the NEXT complete database version. Secondly, to be notified whenever a complete version is reached, using a continuous operator. Listing 3.7 shows the first case.

3.5 Implementation

Now, as the sandbox has been formally specified, and an accompanying SQL syntax has been introduced and applied to use cases, we will show algorithms to compute completeness for sandbox instances. First, we explain our proposed sandbox workflow. Secondly, we discuss when and how completeness checking can be done and introduce a very basic algorithm for completeness checking. Finally, we show how parts of this approach can be optimized under different constraints. The focus of this work lies on completeness checking and the optimization thereof.

In this work, we have decided to restrict the expressive power of completeness conditions in order to allow certain optimizations in the implementation.

We allow users to perform selection and join operations plus quantification (existential and universal ones). Selection and join are the most important building blocks of relational algebra.

Once the characteristics of these operators are known, they could be used to predict the behavior of more complex operations (this lies outside of the scope of this work, though).

So we accept a trade-off of expressive power, thereby gaining an efficient implementation that can be used to evaluate different approaches of completeness checking.

3.5.1 Workflow

In this section we propose a workflow that could be used to implement the sandbox concept in a relational database. To create a sandbox, the database only needs to store the sandbox definition.

1. The user creates a sandbox definition to define the completeness predicate and relation predicates.
2. The database validates and stores the sandbox definition.

When a user queries a sandbox, the workflow is more complicated.

1. The user issues a command to create a sandbox instance, possibly binding parameter values.
2. The database checks if the start time has been reached already; if not it waits until the time has been reached. By default it does not wait as the default start time is the current time.
3. For each committed transaction the database tests if the completeness predicate is true. If so, it executes the user's instructions and then destroys the sandbox instance.

An exception to this approach are continuous sandboxes. In this case, the user asks the database for complete versions and queries those versions in a second operation. Although we do not specify a syntax for it, we still provide the algorithm to implement it (see 3.5.4 on page 25).

1. The user issues a command to create a continuous sandbox instance.
2. The database checks if the end of the time span has been reached already; if not it waits until the time has been reached.
3. For each committed transaction the database tests if the completeness predicate is true. If so, it notifies the user about the complete version, for example by providing a time stamp. In addition to that, it remembers that another complete version has been found.
4. If the end of the time span is reached, or enough complete versions have been encountered (if a limit was defined by the user), the database destroys the sandbox instance.

3.5.2 When to check for for completeness

As discussed in the theory section (3.2.3), the sandbox concept requires distinct transactions. Only between transactions completeness can be reached, but complete versions occurring within a transaction that do not materialize are irrelevant.[3]

The algorithms we propose take this into account. For every transaction, there exist exactly two points in time when completeness can be computed. The first option is to compute completeness at the end of a transaction after the user has made all changes and the database is about to commit the changes. The second option is to compute completeness after a transaction has been committed. In this work we focus on checking completeness at the end of a transaction, using the first option. The second approach is semantically equivalent but has different implementation challenges. The main problem with inter-transaction completeness checking is that certain versions might be lost for computing completeness, because transactions run concurrently. Due to this, there is no defined inter-transaction state. On one hand, this could lead to incorrect results. On the other hand, it makes some optimizations impossible to implement.

Note that we introduce techniques that allow to delay completeness checking to a later point in time, which could be used to overcome this issue. It is based on temporal features, which are explained in 3.7.1 on page 33.

[3] For example, consider a transaction that inserts a value and immediately deletes it again, so that it never materializes in the database. Even if that tuple would have contributed to completeness, it never has been visible from outside of the transaction, so it is irrelevant.

In any case, implementations will have to ensure that completeness checking does not alter the behavior of the database, such as causing more rolled back transactions. However, it might be acceptable to increase the time a transaction needs to complete (cf. 3.2.5).[4]

To sum it up, we have three choices for completeness checking: First, to check at the end of a transaction, guaranteeing that all versions are checked immediately. Secondly, to check after transactions, which could lead to lost versions. Thirdly, to check after transactions are complete using a temporal table logging transaction's changes, guaranteeing that all versions are checked but at the price of increased storage requirements and with possibly longer response times between reaching completeness and notifying about it.

3.5.3 Intra-transaction completeness checking

Working with a transaction typically consists of at least two phases: First, it is used to access, modify, insert and delete data. Secondly, it is committed. The result of a commit can either be a rollback or, if successful, all modifications are stored. The intra-transaction completeness checking approach tests the database for completeness after the user made all changes and before the database commits. It intercepts commits and executes the completeness checks within the transaction before the database commits it. At this point in time it is safe to assume that no changes by other transactions are visible if the database provides snapshot isolation. Note, though, that this approach might slow down user transaction processing (cf. 4.8.1).

Snapshot isolation In 3.2.5 we defined the requirements of sandboxes and noted that sandboxes must not cause additional rollbacks compared to a database without sandboxes. This requirement can only be fulfilled by snapshot isolation.

Assume that a database uses two-phase locking to prevent concurrent updates of data and to avoid deadlocks. The notation used here distinguished between read (r) and write (w) operations on a specific datum. This is important because for the first occurrence of a datum within a transaction a lock is acquired. There are two concurrent transactions, T_1 and T_2, each writing a different datum:

$$T_1 = w(a)$$
$$T_2 = w(b)$$

Both transactions can be executed as there is no conflict between them. Now, there exists a sandbox with a completeness constraint requiring both values a and b, executed within the two transactions. The two transactions now become:

$$T_1 = w(a)r(a)r(b)$$
$$T_2 = w(b)r(a)r(b).$$

[4] This is not entirely true because longer running transactions could cause more rollbacks, a fact that has to be taken care of.

When executed concurrently, both transactions perform their write operation, thus acquiring locks on a and b. During the completeness checking, each transaction also needs to access the datum locked by the other transaction, leading to a circular data dependency, which is a deadlock. Here, the database would rollback one of the transactions, but this is not desired. This kind of problems can only be solved correctly using snapshot isolation. In this isolation level the database appears to take a snapshot at the beginning of a transaction. Thus, $r(a)r(b)$ would be executed on data that was in the database at the beginning of the transaction if not locally updated.

3.5.4 Naïve algorithm

The naïve approach is to check for completeness just before a transaction commits, using the intra-transaction completeness checking approach. A database needs to provide the hook for this operation to be injected. For the naïve implementation, we assume the database provides an interface to register transaction callbacks. All following snippets are based on Java-like code.

We call this implementation *naïve* (in contrast to more sophisticated approaches) because it checks completeness for each version instead of combining knowledge gained from a previous version with the current version. In section 3.8, we present different approaches, which still have the naïve approach in mind but optimize certain operations. We are using a naïve approach as it best shows the desired behavior.

For convenience we are using an abstract `Version` type to denote a monotonically increasing logical clock. In a real-world implementation, this could be a sequence number, a transaction identifier, or a real time value.

Database and sandbox interfaces

The database has to provide a set of functions in order to implement completeness checking. Most importantly, it has to offer some kind of hook into the transaction processing system. For now, we assume that there exists such a function. In a real database, this might not exist and would need to be implemented, either inside the database or using a middleware approach.

Listing 3.8 shows the interface the database implements. It provides a specialized observer pattern, a software design pattern to observe another object for defined events, such as state changes or occurring events. In this case, we use it to hook into the transaction processing system. The database will invoke all registered callbacks before committing a transaction. In our simplified model, the database also exposes the current version identifier, and allows to run a query on a specific version of the database. This is comparable to the time-sliced query syntax in SQL:2011 [9].

Listing 3.8: Database definition

```
1 interface DB {
2     void register(SandboxInstance);                    ▷ Register a callback
3     void unregister(SandboxInstance);                  ▷ Unregister a callback
4     Version currentVersion();                          ▷ Returns current head version
5     Result query(Query, Version);                      ▷ Queries a database version
6 }
```

```
1 class SandboxDefinition {
2     String name;                                       ▷ Sandbox name
3     Query Qs;                                          ▷ Rewritten completeness constraint $p_s$
4     Map<Relation, Query> Qr;                           ▷ Rewritten visibility constraint set $P_r$
5     Result query(Query, Version);                      ▷ Query the sandbox
6 }
```

The database stores sandbox definitions, consisting of a name, a completeness predicate and relation predicates. For clients, a sandbox definition also provides a query method to query a specific version of the database while applying the relation predicates. A sandbox definition has the structure shown in listing 3.9, which directly corresponds to the SQL syntax to create a sandbox shown in 3.3.1. Listing 3.9 shows the data stored by a sandbox definition in pseudo-code.

Sandbox instances

While the sandbox definition is very close to the CREATE SANDBOX command defined in 3.3.1, creating a sandbox instance is somewhat different. For a sandbox instance, we map the mode setting to a concept covering all operators instead of having different instances for each concept. The temporal completeness operators are expressed using a time span and a number of expected and encountered complete versions, using the mapping described below. The time span is expressed as versions numbers, consisting of a start and end version number. The end is non-inclusive. The pseudo-code showing the sandbox instance interface is shown in listing 3.10.

Now The NOW operator is equivalent to checking for the next complete version with a minimal time span, e.g. the current time plus the smallest time unit that is non-zero. Translated to a version, this means the current version as start and the next version number as the end of the checked time span. The user is interested in exactly one complete version, or none if the current version is not complete.

Listing 3.10: Sandbox instance

```
1  class SandboxInstance {
2      SandboxDefinition sandbox;                    ▷ Reference to sandbox
3      Version start;                                ▷ First version to consider
4      Version end;                                  ▷ Last version to consider
5      int expectedCS;                               ▷ Number of expected complete versions
6      int encounteredCS;                            ▷ Number of encountered complete versions
7
8      process();                                    ▷ Process request
9      callback(Version currentVersion);             ▷ Called by the database
10     checkVersion(Version i);                      ▷ Test and act on complete version
11 }
```

Next The NEXT operator is equivalent to checking for a single complete version either using the default time span (now until indefinite future) or a user provided time span.

Last The LAST operator is equivalent to looking for one complete version beginning at backwards from the end of the time span, which can be user defined. The default time span starts at the first database version and ends at the current version.

Whenever a user queries data within a sandbox, the database creates a sandbox instance using the user-provided temporal completeness operator. Also, we use a counting approach to remember how many complete versions the user requested and how many have been encountered so far. The sandbox instance pseudo-code is shown in listing 3.10.

Naïve completeness checking

The naïve algorithm checks for completeness after every transaction without leveraging any prior knowledge or sharing between sandboxes. It conceptually consists of two parts. The first part is responsible for checking existing data in the database, i.e. past data, while the second part is listening for database callbacks and taking appropriate actions. The database and the completeness algorithm implement the observer pattern. Listings 3.11 and 3.12 show the two algorithms. Here it can be seen that the naïve approach tests each database version individually for completeness by iterating over individual versions within the provided time span.

NEXT processing for past data just iterates over existing versions of the database and checks each individually. For processing of new data, it uses the database callback to test each new version individually. Past processing always waits until the database version is

Listing 3.11: SandboxInstance processing

```
1  public void process() {
2      if (start <= end) {                          ▷ Now, next, Continuous processing
3          for (int i = start; i <= Math.min(end, DB.currentVersion()); i++)
               {                                                ▷ Processing previous versions
4              if (expectedCS <= encounteredCS) {
5                  return;                          ▷ Encountered enough complete versions
6              }
7              checkVersion(i);                                          ▷ Test version i
8          }
9          if (expectedCS > encounteredCS) {          ▷ Are more versions requested?
10             if (end <= DB.currentVersion()) {
11                 notifyClient(end, false);
12             } else {
13                 DB.register(this);                  ▷ Register this SandboxUsage
14             }
15         }
16     } else {                                                ▷ Processing Last operator
17         if (start > DB.currentVersion()) {
18             DB.register(this);                  ▷ Wait until end has been reached...
19         } else {
20             int i = start;            ▷ Iterate over versions from new to old as needed
21             while (i >= end && encounteredCS < expectedCS) {
22                 checkVersion(i);
23                 i--;
24             }
25             if (encounteredCS < expectedCS) {
26                 notifyClient(end, false);
27             }
28         }
29     }
30 }
```

Listing 3.12: SandboxInstance callback

```
1  public void callback(int currentVersion) {
2      if (start <= end) {                          ▷ Now, next, continuous processing
3          if (currentVersion >= end) {                          ▷ Do we have to stop here?
4              notifyClient(end, false);
5              DB.unregister(this);
6          }
7          if (start <= currentVersion) {
8              checkVersion(currentVersion);
9              if (encounteredCS >= expectedCS) {
                                        ▷ Did we encounter enough complete versions?
10                 DB.unregister(this);
11             }
12         }
13     } else {                                                  ▷ Processing last operator
14         if (start <= currentVersion()) {
15             DB.unregister(this);                          ▷ No more callbacks required.
16             int i = start;            ▷ Iterate over versions from new to old as needed
17             while (i >= end && encounteredCS < expectedCS) {
18                 checkVersion(i);
19                 i--;
20             }
21             if (encounteredCS < expectedCS) {
22                 notifyClient(end, false);
23             }
24         }
25     }
26 }
```

Listing 3.13: Sandbox check version

```
1 public void checkVersion(Version i) {
2     Result = DB.query(sandbox.Qs, i);                    ▷ Run p_s on database version i
3     if (result.size() > 0) {                                          ▷ Is p_s satisfied?
4         encounteredCS = encounteredCs + 1;
                                        ▷ Update encountered complete versions counter
5         notifyClient(i, true);   ▷ Notify the user a complete version has been reached.
6     }
7 }
```

Listing 3.14: Querying a sandbox

```
1 public Result query(Query q, Version i) {
2     for (Relation r : q.relations()) {
                                  ▷ Replace references to relations in P_r by their selection
3         if (sandbox.Qr.contains(r) {
4             q = q.replace(r, sandbox.Qr.get(r));
5         }
6     }
7     return DB.query(q, i);                                ▷ Run adapted q on version i
8 }
```

equal to or past the end of the specified time span. At this point, it will iterate over past versions in reverse, also checking each version separately.

Checking a single version for completeness The naïve implementation requires a function to check a single database version for completeness. It takes a sandbox definition and a version to compute completeness. If the version is found to be complete, it notifies the client. The pseudo-code is shown in listing 3.13.

Rewriting queries applied on a sandbox Clients can query the database through a sandbox. When an implementation chooses to use a middleware approach, queries for that sandbox need to be rewritten such that all relations are replaced by the filtered relations obtained from applying P_r. There are several possibilities to achieve this, for example using views or materialized views or by adding the constraints to the query directly. Here, we conceptually choose to replace references to relations in the database by a selection of this relation, e.g. a query based on T would be rewritten to use $\sigma_{p_T}(T)$ instead. The pseudo-code for this function is shown in listing 3.14.

3.6 Cost model and evaluation

The cost of computing completeness depends on several different implicitly defined parameters in the algorithms of section 3.5. The analysis aims at providing an upper bound of time complexity of those algorithms.

Querying a database This is the cost associated with executing a single query on a specific version of the database. We consider the database to be a black box and not part of our application, so we cannot give a more specific cost analysis than this. However, there is a constant cost associated with preparing the query execution plus a variable factor that depends on how much data is accessed and put out by the database. Thus, we characterize it as a constant (C) plus a variable part.

$$\mathsf{cost}(\text{DB.query}(q)) = C + \mathsf{sel}(q) \tag{3.19}$$

Checking version The cost of checking a single database version for completeness is dominated by the cost of executing the completeness predicate p_s, or more precisely, the transformed query Q_s. Thus, it is equal in magnitude to the cost of querying the database.

$$\mathsf{cost}(\text{checkVersion}) = \mathsf{cost}(\text{DB.query}) \tag{3.20}$$

Completeness usage time span Each sandbox instance has a start and end value. The upper bound of completeness checks, or versions in that time span, is denoted by I, which is

$$I = |\text{start} - \text{end}|. \tag{3.21}$$

3.6.1 Continuous queries

To aid our analysis, we only consider continuous queries because all other temporal completeness operators can be reduced to a continuous query. Continuous queries check every version of a specific time span for completeness, not taking into account the number of encountered complete versions. Thus, I is the number of versions the algorithm has to work on.

Single sandbox instance

Assume that there exists a single sandbox instance operating in continuous mode. So, the expected number of complete versions is equal to or bigger than I. Using this, the cost of running the completeness check for the sandbox instance is

$$\mathsf{cost}(SBI) = \mathsf{cost}(\text{DB.query}) \cdot I \tag{3.22}$$

Single sandbox, multiple instances

In this case we assume a single sandbox of which exist multiple sandbox instances. The naïve implementation does not combine completeness checking for several sandbox instances and might execute the same query several times per version. Assuming there are U sandbox instances, the cost is

$$\begin{aligned} \mathsf{cost}(MSBI) &= \mathsf{cost}(SBI) \cdot U \\ &= \mathsf{cost}(\text{DB.query}) \cdot I \cdot U \end{aligned} \tag{3.23}$$

One observes that all sandbox instances using the same sandbox also share a common p_s represented by Q_s in the naïve implementation. The result of Q_s only depends on the database version it is executed on, but does not change when executed multiple times on the same version. Thus, it only needs to be executed once per version.

$$\mathsf{cost}(MSBI) = \mathsf{cost}(SBI) \tag{3.24}$$

$$= \mathsf{cost}(\text{DB.query}) \cdot I \tag{3.25}$$

Multiple sandboxes

Let S be the number of distinct sandboxes in the system. Two sandboxes are distinct if their p_s are different. The naïve implementation evaluates each sandbox instance individually, resulting in a cost of

$$\begin{aligned} \mathsf{cost}(MSB) &= \mathsf{cost}(MSBI) \cdot S \\ &= \mathsf{cost}(\text{DB.query}) \cdot I \cdot S \end{aligned} \tag{3.26}$$

3.6.2 NEXT and LAST temporal completeness operators

The cost analysis for the NEXT and LAST temporal completeness operators is very similar. As we do not know anything specific about the data, we cannot make any assumption on the shape of the completeness function's output.

3.6.3 NOW queries

The NOW temporal completeness operator behaves differently, because we know it is only executed a single time, removing the factor I encountered in the analysis of continuous and NEXT/LAST temporal completeness operators.

In conclusion, the performance of computing completeness version by version is close to constant for each individual version and linear for an interval.

3.7 Required database features

In the previous sections we introduced several features our system should support. Our goal is to implement those features on a regular relational database, using mostly SQL to express operations. In order to provide access to past versions of the database, the database system has to maintain a history of all changes. The most common, and, since SQL:2011 [9], standardized technique is to have temporal tables with system time. Some of our approaches also depend on database changes of the currently executed transaction to be available in a special changelog table. Some database vendors include part of this functionality in their software already, but as it is not commonly available, we decided to implement it ourselves.

3.7.1 Temporal tables

A temporal table is an extension to regular tables that has a system time attribute added to the relation. The system time describes when a tuple came into existence and when it was removed. There are several possibilities to implement this feature on standard databases. In the context of this work we use a simple variant of regular tables: shadowed tables that contain the full history of a particular table. This is similar to the design proposed by [10], only that we do not use a specifically optimized index.

For a relation with three attributes, e.g. $R(a, b, c)$, there exists a temporal relation $R^T(a, b, c, s, e)$ where s is the time a tuple came into existence, and e when it was removed. For tuples that are not yet deleted, the end version is an infinite value.

3.7.2 Changelog tables

For some computations it might be required to store changes of a transaction in a separate table. For this we introduce the concept of a changelog table, a table that stores another table's tuples that were deleted and/or inserted during a transaction.

Given a relation $R(a, b, c)$, the changelog table would have the structure $R^C(a, b, c, \delta)$ where δ is the event type, which is either insertion or deletion. Updates to a tuple are treated as deleting the old tuple and inserting the new one.

3.8 Optimization of completeness checking

The analysis of the naïve implementation has shown that the main cost is associated with running a sandbox's completeness predicate using `DB.query`. We propose different optimization approaches, which are later to be verified. In general, we want to optimize time and not space complexity. Many approaches towards the optimization of databases

already exist. We will shortly describe some common concepts and discuss their usefulness towards the optimization of completeness checking.[5]

Database optimizations The standard approach for relational databases is to optimize queries using optimized database features, e.g. indices, better query execution plans, etc. Other optimizations include elimination of common subexpressions, query rewriting, automatic view materialization, caching and so on.

We assume a database applies those optimizations, but this is not obvious, as we consider the database a black box. It is outside of the scope of this work to analyze each intra-database optimization and its effect on completeness checking, as this is not fundamentally different to optimizing regular SQL queries.

Incremental computation Incremental view maintenance has been around for many years as an optimization to keeping views up to date by incorporating changes performed by a transaction. The idea is to rewrite queries such that they compute the delta of their result based on the delta of input relations. As deltas are considered to be small, this could provide a huge performance improvement. In the case of completeness checking, this can be utilized and further optimized as, according to 3.2, we are only interested in the cardinality of a set, not the actual contents.

Temporal computation An orthogonal approach to checking completeness using the database's contents or changes at the end of a transaction is to run completeness checks on the data history. This requires data history to be available as a temporal table. This approach can be used to compute completeness of single versions, but also of a range of versions.

As specified in the theory section (3.2), completeness predicates can be expressed by relational algebra formulas with a quantification operator. Hence, we can try to optimize the relational algebra operation to get a faster operation that gives the same result. Databases perform those kind of operations, but, as we impose additional restrictions on the completeness predicates we have more knowledge about the data. We also know how changelog and temporal tables are behaving. Thus, we are able to apply more optimizations.

3.8.1 Incremental processing

Incremental processing means that the result of a query is partially computed based on changes of a transaction. We use a notation based on [7] and [21] that expresses the previous version of an operation, the delta of the actions it performed and the version after applying those changes.

[5] A detailed list of related work and techniques can be found in chapter 2 on page 9.

- pre is a function that returns a relation before executing a transaction.
- pst is a function that returns a relation after executing a transaction (*post*).
- dlt is a function that returns the difference between pre and pst (*delta*). It holds that

$$\mathsf{pst}(R) = \mathsf{pre}(R) \uplus \mathsf{dlt}(R). \tag{3.27}$$

  The $\uplus$ operator applies the changes of the delta to the relation on the other side. The delta contains both inserted and deleted tuples. We define the $\uplus$ operator as applying a delta in reverse:

$$\mathsf{pre}(R) = \mathsf{pst}(R) \uplus \mathsf{dlt}(R). \tag{3.28}$$

  Both operators have a lower precedence than join operators.

- The functions pre, pst and dlt are commutative with selection:

$$\sigma_\varphi(\mathsf{pre}(R)) = \mathsf{pre}(\sigma_\varphi(R)) \tag{3.29}$$
$$\sigma_\varphi(\mathsf{pst}(R)) = \mathsf{pst}(\sigma_\varphi(R)) \tag{3.30}$$
$$\sigma_\varphi(\mathsf{dlt}(R)) = \mathsf{dlt}(\sigma_\varphi(R)) \tag{3.31}$$
$$\tag{3.32}$$

Using this notation we can show how relational algebra expressions can be transformed to incremental processing. Note that the *pst* of an operation must be the same for incremental and non-incremental computation.

Selection Selection is a simple operation that generates a new relation from an input relation retaining those elements that match a predicate. Let R be an input relation and φ a predicate. Then,

$$S = \sigma_\varphi(R). \tag{3.33}$$

Applying incremental operations we derive the following equation:

$$\begin{aligned} \mathsf{pst}(S) &= \mathsf{pst}(\sigma_\varphi(R)) \\ &= \sigma_\varphi(\mathsf{pst}(R)) \\ &= \sigma_\varphi(\mathsf{pre}(R) \uplus \mathsf{dlt}(R)) \\ &= \sigma_\varphi(\mathsf{pre}(R)) \uplus \sigma_\varphi(\mathsf{dlt}(R)) \end{aligned} \tag{3.34}$$

Join For joins, a similar transformation is possible. Let P and R be relations and S their inner join. Then,

$$S = P \bowtie R. \tag{3.35}$$

Applying incremental operations we derive the following equality:

$$\begin{aligned} \mathsf{pst}(S) &= \mathsf{pst}(P \bowtie R) && (3.36)\\ &= \mathsf{pre}(S) \uplus \mathsf{dlt}(P \bowtie R) \\ &= \mathsf{pre}(S) \\ &\quad \uplus \mathsf{pre}(P) \bowtie \mathsf{pre}(R) \uplus \mathsf{pre}(P) \bowtie \mathsf{dlt}(R) \\ &\quad \uplus \mathsf{dlt}(P) \bowtie \mathsf{pre}(R) \uplus \mathsf{dlt}(P) \bowtie \mathsf{dlt}(R) \\ &= \mathsf{pre}(S) \\ &\quad \uplus \mathsf{pst}(P) \bowtie \mathsf{dlt}(R) \uplus \mathsf{dlt}(P) \bowtie \mathsf{pst}(R) && (3.37)\\ &\quad \uplus \mathsf{dlt}(P) \bowtie \mathsf{dlt}(R) \end{aligned}$$

3.8.2 Incremental processing of the completeness predicate/counting

Our goal is to apply incremental processing when counting the size of the sets. Without loss of generality we use completeness predicates of the form (see formula 3.17)

$$|S| \; op \; n$$

where S is some relation, op is a comparison operator and $n \in \mathbb{N}$ a natural number. We know that the comparison of two numbers is fast, so we need to improve the performance of the count operator to determine the cardinality of a set efficiently. For general counting, $s = |S|$, we can use the following approach:

$$\begin{aligned} s &= |S| \\ &= \mathsf{count}(S) \\ &= \mathsf{count}\left(\mathsf{pre}(S) \uplus \mathsf{dlt}(S)\right) \\ &= \mathsf{count}(\mathsf{pre}(S)) + \mathsf{count}(\mathsf{dlt}(S)) \end{aligned} \tag{3.38}$$

Note that the count of dlt can be negative, while the count of pre and pst is always positive or zero. The values of $\mathsf{pst}(S)$ and $\mathsf{dlt}(S)$ are known at the end of a transaction, but not necessarily the value of $\mathsf{pre}(S)$. As we are interested in the cardinality of the query's result, it is enough to remember the old count at the beginning of a transaction.[6]

An efficient incremental computation is one where the innermost operator is one of pre, pst or dlt. This is because the sets of data before or after a transaction are known as well as the delta. Also, every formula can be transformed into one that either depends on pre or pst, but not both. This eliminates the need of the database to preserve the before-state until the end of a transaction, as seen in formula 3.37.

[6] For concurrent transactions a different approach might need to be used.

Selection The counting selection operator can be transformed to an incremental variant using the following transformation. Let $s = |\sigma_\varphi(R)|$.

$$\begin{aligned} s =& |\sigma_\varphi(R)| && (3.39)\\ =& \mathsf{count}(\sigma_\varphi(R)) \\ =& \mathsf{count}(\sigma_\varphi(\mathsf{pre}(R) \uplus \mathsf{dlt}(R))) \\ =& \mathsf{count}\left(\sigma_\varphi(\mathsf{pre}(R)) \uplus \sigma_\varphi(\mathsf{dlt}(R))\right) \\ =& \mathsf{count}(\sigma_\varphi(\mathsf{pre}(R))) + \mathsf{count}(\sigma_\varphi(\mathsf{dlt}(R))) && (3.40) \end{aligned}$$

The resulting operation only depends on the previous result and the result computed over the delta, thus it is efficient according to our initial definition.

Join Equivalently, the join operator can be transformed into an incremental counting variant. Using above formulas and $s = |P \bowtie R|$, we get:

$$\begin{aligned} s =& |P \bowtie R| && (3.41)\\ =& \mathsf{count}(P \bowtie R) \\ =& \mathsf{count}(\mathsf{pre}(P \bowtie R) \uplus \mathsf{dlt}(P \bowtie R)) \\ =& \mathsf{count}\Big(\mathsf{pre}(S) \\ & \uplus \mathsf{pre}(P) \bowtie \mathsf{pre}(R) \uplus \mathsf{pre}(P) \bowtie \mathsf{dlt}(R) \\ & \uplus \mathsf{dlt}(P) \bowtie \mathsf{pre}(R) \uplus \mathsf{dlt}(P) \bowtie \mathsf{dlt}(R)\Big) \\ =& \mathsf{count}(\mathsf{pre}(S)) \\ & + \mathsf{count}(\mathsf{pst}(P) \bowtie \mathsf{dlt}(R)) + \mathsf{count}(\mathsf{dlt}(P) \bowtie \mathsf{pst}(R)) && (3.42)\\ & - \mathsf{count}(\mathsf{dlt}(P) \bowtie \mathsf{dlt}(R)) \end{aligned}$$

We can see in formula 3.42 that the cardinality of the join result can be incrementally computed. It does not need to be computed completely, only the changed rows have to be used to update the count. This is because $\mathsf{count}(\mathsf{pre}(S))$ is known from the previous version.

3.8.3 Temporal processing

Another approach to computing completeness is by using temporal features of a database in order to access past versions. The temporal features are required to be able to execute queries on past versions of the database and store tuples together with a system time attribute. The system time attributes are logical time stamps indicating when a tuple came into existence and when it was removed. Using this, we can formulate queries that check whether a certain version is complete, and also, in a second step, if a range of versions is complete.

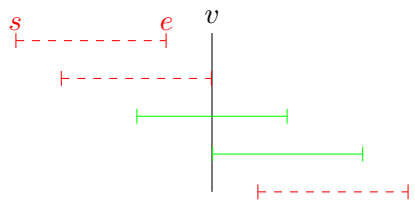

Figure 3.2: Visualization of a temporal single version selection. Each line represents a tuple. Only tuples marked with a solid line are included in the output.

3.8.4 Single version completeness checking

Checking a single version for completeness is very similar to checking a regular table for completeness. A regular table contains only data visible in the latest version of the database, but no previously deleted data. To run a completeness check, we select those tuples from a temporal table that are visible at a given version.

Let R^T be a temporal table. According to section 3.7.1 on page 33 it has additional attributes s and e when comparing it to the relation R. s is the inclusive start time of a tuple and e the non-inclusive end time of a tuple, e.g. it is visible in version s but not anymore in version e. Given a version v, we can reconstruct the original relation at time v using

$$R_v = \sigma_{s \leq v \wedge e > v}(R^T). \tag{3.43}$$

Selection The relation of a specific version can be plugged into a regular completeness predicate. If the selection predicate is $\sigma_\varphi(R)$ with version unspecified, we can use the above approach to turn it into a selection predicate operating on a specific version:

$$\begin{aligned} \sigma_\varphi(R_v) =& \sigma_\varphi(\sigma_{s \leq v \wedge e > v}(R^T)) && (3.44)\\ =& \sigma_{\varphi \wedge s \leq v \wedge e > v}(R^T) && (3.45) \end{aligned}$$

Join As a relation at a specific version represents exactly the data that was visible at this particular point int time, we can use it directly to compute joins. Let v again be a version. Then,

$$T_v \bowtie U_v = \pi_T(\sigma_{s \leq v \wedge e > v}(T^T)) \bowtie \pi_U(\sigma_{s \leq v \wedge e > v}(U^T)) \tag{3.46}$$

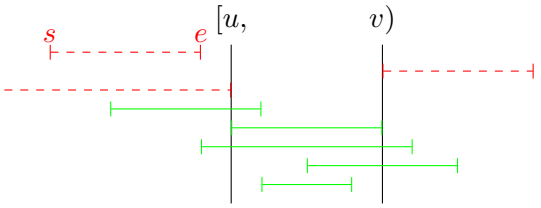

Figure 3.3: Visualization of a temporal range version selection. Each line represents a tuple. Only tuples marked with a solid line are included in the output.

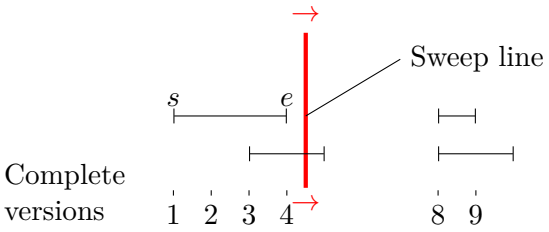

Figure 3.4: Visualization of a temporal range sweep line approach. Each horizontal line represents a tuple, the vertical line is the sweep line. Below the tuples complete versions are shown.

3.8.5 Version range completeness checking

A logical extension to checking a single version using temporal data is to check a range of versions. Again, let R^T be the temporal table for table R and u and v versions, such that the range $[u, v)$ is to be checked for completeness. The database can only return us tuples that were visible at some point in the specified time span, but it cannot directly tell us which versions meet the completeness constraints given by a specific sandbox definition. It hard (if at all possible) to express this using SQL, so we use additional software to implement this feature. The following expression selects all tuples that are visible inside the specific time span $[u, v)$. Figure 3.3 visualizes this.

$$R_{[u,v)} = \sigma_{s<v \wedge e>u}(R^T) \tag{3.47}$$

In a second step the result needs to be ordered by s and e. Then, a sweep line approach can be used to iterate over complete versions in the database. The sweep line approach is visualized in figure 3.4.

Selection As it is the case for single version completeness checks, we can just plug in the resulting relation $R_{[u,v)}$ into the completeness predicate σ_φ:

$$\sigma_\varphi(R_{[u,v)}) = \sigma_\varphi(\sigma_{s<v \wedge e>u}(R^T)) \tag{3.48}$$
$$= \sigma_{\varphi \wedge s<v \wedge e>u}(R^T) \tag{3.49}$$

Join Joining two version ranges is not as simple as joining single versions. The issue is that upon joining we have to ensure that the tuples being joined actually exist at the same point in time, i.e. there exists a version where both sides are 'alive.'

$$T_{[u,v)} \bowtie U_{[u,v)} = \sigma_{s<v \wedge e>u}(T^T) \underset{\substack{T.s \leq U.s \wedge U.s < T.e \\ \vee\ U.s \leq T.s \wedge T.s < U.e}}{\bowtie} \sigma_{s<v \wedge e>u}(U^T) \tag{3.50}$$

3.8.6 For-all counting approach

The for-all operator introduced in the SQL extension section as shown in section 3.3.3 on page 19 can be implemented using a counting approach similar to the above optimizations. For-all queries have the following form:

$$\forall x \in R : p(x). \tag{3.51}$$

This is equivalent to the following expression (cf. 3.2.3):

$$0 = |R| - |\sigma_p(R)|. \tag{3.52}$$

Using a counting approach, we can transform the equation to an incremental variant:

$$\begin{aligned} 0 &= |R| - |\sigma_p(R)| && (3.53)\\ &= \mathsf{count}(R) - \mathsf{count}(\sigma_p(R)) \\ &= \mathsf{count}(\mathsf{pre}(R) \uplus \mathsf{dlt}(R)) \\ &\quad - \mathsf{count}(\mathsf{pre}(\sigma_p(R)) \uplus \mathsf{dlt}(\sigma_p(R))) \\ &= \mathsf{count}(\mathsf{pre}(R) \uplus \mathsf{dlt}(R)) \\ &\quad - \mathsf{count}(\sigma_p(\mathsf{pre}(R)) \uplus \sigma_p(\mathsf{dlt}(R))) \\ &= \mathsf{count}(\mathsf{pre}(R)) + \mathsf{count}(\mathsf{dlt}(R)) \\ &\quad - \mathsf{count}(\sigma_p(\mathsf{pre}(R))) + \mathsf{count}(\sigma_p(\mathsf{dlt}(R))) && (3.54) \end{aligned}$$

The above expression can be computed incrementally because the last transaction's counts can be remembered and the difference to the new count can be calculated from the transaction changelog.

4 Benchmark

The benchmark is used to compare different approaches and verify our performance assumptions. We measure the response time for different completeness operator implementations. We assume that the naïve approach is not good in all cases, and it is in general better to compute completeness using less data, which can be achieved by using changelog tables. We compare selection, join and for-all operators for live data and selection and join for past database versions. The benchmark is implemented as a middleware intercepting commit instructions for the database. It is written in Java, using Java Database Connectivity (JDBC), and runs on Postgres. We use it to measure the performance of the different approaches of testing if a certain database version is complete. For this reason, we structure each transaction in the following distinct steps:

1. Prepare the database for the transaction. This includes clearing the changelog table and saving a new time stamp plus transaction identifier in the database.
2. Run the user's transaction while recording changes to the database.
3. Run each completeness operator.

For each of these operations we measure the time it takes to get a response from the database plus processing the result locally.

The workload for our benchmark is based on the Transaction Processing Performance Council C benchmark (TPC-C)[1]. TPC-C is a standard describing a workload to test the transaction performance of databases [1]. We use an open-source implementation and run it on top of our middleware software.

4.1 TPC-C

Each TPC-C update performs one of the following actions (the probability for each action is shown in percent as defined in the TPC-C benchmark):

Payment An order will be payed, by adding the relevant details to the database. This includes writing to the database (43%).

[1] The TPC-C benchmark is used to evaluate the transaction performance of databases by simulating a order/warehouse model with a generated workload

Listing 4.1: TPC-C selection-based sandbox

```
1  CREATE SANDBOX low_stock(IN :warehouse int, IN :item_id int)
2  WHEN
3      EXISTS (SELECT * FROM stock WHERE s_w_id = :warehouse AND
4          s_i_id = :item_id AND S_QUANTITY < 100)
5  WITH
6      warehouse w: w.w_id = :warehouse,
7      stock s: s.s_i_id = :item_id,
8      oorder o: o.o_i_id IN (
9          SELECT o_i_id FROM oorder o
10         INNER JOIN order_line ol ON ol.OL_O_ID = o.OL_O_ID
11         WHERE ol.OL_I_ID = :item_id
12             AND ol.OL_W_ID = :warehouse);
```

Stock level Check the stock level (4%). This is a read only transaction.

Order status Check the order status of a transaction (4%). This is a read only transaction.

Delivery Updates an order to reflect the delivery status (4%). This involves reading, updating and deleting tuples from the database.

New order Place a new order for some items in a warehouse (45%).

4.1.1 Sandboxes

We define three sandboxes for the benchmark, one based on selection and another one based on a join. A third sandbox is based on a for-all query.

Selection sandbox

The selection sandbox is based on the stock relation and checks for items with low stock. It takes a warehouse and an item as a parameter and is complete if the count (an attribute) of that item is below 100. To the user it exposes just the chosen warehouse and stock, plus all orders that include the particular item from the warehouse. The sandbox definition is shown in listing 4.1.

Listing 4.2: TPC-C join-based sandbox

```
1 CREATE SANDBOX irregular_client
2 WHEN
3     EXISTS (SELECT * FROM order_line ol INNER JOIN order o
4         ON o.O_ID = ol.O_ID
5         WHERE o.o_ol_cnt = 5 AND ol.ol_amount > 9000);
```

Listing 4.3: TPC-C for-all sandbox

```
1 CREATE SANDBOX local_orders
2 WHEN
3     FORALL ( SELECT * FROM orders o)
4     SATISFY ( o.O_ALL_LOCAL = 1 );
```

Join sandbox

The join sandbox is used to check for irregular clients, which are defined as having orders with few items but a large amount of those items. It is not parameterized. Inside the sandbox, it exposes all data. The sandbox definition is shown in listing 4.2.

For-all sandbox

To test the for-all operator, we define a sandbox that is complete when all items of an order come from the same district. It is shown in listing 4.3.

4.2 Operators

For the benchmark, we implemented selection and join operators as well as a for-all operator in SQL. We distinguish between three different operator types:

Basic A basic operator is an SQL expression working on regular tables. It can be used to check whether the current version is complete or not at the end of a transaction. The basic operator does not use the changelog or a temporal table to compute its result.

Incremental An incremental operator is executed on different database versions in sequence, in the order of a serialized history. The incremental operator can obtain changes either from the changelog table or the temporal features a database offers.

	Table	changelog	Temporal table
Basic	I		
Incremental		I	I/P
Temporal			I/P

Table 4.1: Operators and tasks. Rows show operator types, columns database capabilities. The letter I indicates that an operator can run inside a transaction, the letter P means it can run on past data.

If it uses the changelog table it can only be run inside a transaction, because the changelog table would be empty afterwards.

Temporal A temporal operator retrieves data within a time span that contributes to the set of complete data. It does not follow an incremental approach, but receives all relevant data in one database operation. A sweep line approach can be used to compute the count of active tuples at each version.

4.3 Tracking changes: Triggers

SQL is a language to express *commands* of what a database should do with data. The database then executes those commands based on the data it stores. While this is usually intended, we need to be aware of all changes that are performed. This means we can either restrict the statements that are executed to a set with predictable effects, or we hook into the database to be notified about all changes. For our implementation we chose the latter approach and implemented the change tracking inside the Postgres database. We create triggers for all tables in our database reacting to updates, deletes and inserts. The trigger in turn puts the tuple into the changelog table and updates the temporal table accordingly.

4.4 Cost of maintaining temporal and changelog tables

Before testing the performance of individual completeness operator implementations, we look at the overhead incurred by providing the infrastructure for our operators. The infrastructure maintains the changelog table and updates the temporal table reflecting changes performed by a user's transaction. In our implementation, the changelog and temporal table are maintained by triggers that are executed whenever a table is changed. Users do not interact directly with the system-maintained tables. The measured costs are shown in 4.8.1 on page 47.

4.5 Cost of selection completeness operators

We are interested in evaluating the cost of selection operators on both live data and past versions. Selection only tests for the presence of tuples matching a given pattern. We assume it is fast if there is an index that can be queried for existence. If there is no index, selections are generally slow because the database needs to scan through all data. The following list shows our assumptions:

Selection on base table plus index In this case we test for the presence of a tuple with the relevant attributes being included in an index. In this case, responses are expected to be very fast.

Selection on base table without index Here, the database potentially needs to scan all tuples in a table, which is slow. We do not test this case in our benchmark.

Selection on changelog We incrementally compute the count of matching tuples. The difference is computed from the changelog table. As the changelog is supposed to contain a small number of tuples compared to its base table, tests both with and without index are supposed to be fast. This is also related to the distribution of updates as defined in the TPC-C standard. The changelog cannot be used for past versions as it is transient.

Selection on temporal table, single version There are two approaches that fit into this category: Either computing the whole result on that particular version, or just the difference to the previous one. We expect the cost of obtaining tuples from the temporal table to be at least as high as obtaining them from the base table. This is because the temporal table stores at least the data that is stored in the base table and continuously grows with every change.

Selection on temporal table, version range In this approach, we select all matching tuples that intersect with the version range. A sweep line approach is used to count the number of tuples at each version within the version range. The performance of this approach depends on a variety of factors, but it has a benefit of limited database communication.

4.6 Cost of join completeness operators

The operators shown below implement an inner join between two different tables. There are no explicit foreign key relations defined on the database. Our assumptions are described below. In general, joins always join two relations of the same kind. For example, base tables are only joined with other base tables.

Join on base table A join on the base table has the same effect as if the user would be working on a database without temporal and changelog features. The join is recomputed completely on every invocation. We expect this to be slow.

Incremental join on changelog To speed up the processing of the join, we first compute the full join and then incrementally update the result based on changes recorded in the changelog table, joined with the corresponding base table. This is explained in formula 3.42. We expect this to be fast.

Incremental join on temporal table The join can also be incrementally computed using the temporal table, by joining the `AS-OF` version with the changes from the other side of the join, and vice versa. This is essentially the same approach as an incremental join on the changelog, but involves more data and, due to this, is potentially slower.

Single version join on temporal table Joining past data might be required, and this operator joins two tables at a given version. We assume this operator to be slower than joining the base tables.

Range version join on temporal table This approach tries to compute the join on a range of versions, as explained in formula 3.50. Depending on the time span to be covered, this might be worse or better than checking each version for completeness. The actual costs need to be benchmarked.

4.7 Cost of for-all completeness operators

The operators shown below compute the result of a for-all query. We only implemented the for-all operator for live checking because it can be rewritten into an existential operator.

Base for-all The base for-all operator implements a simple approach to check if the base table is complete: It compares the counts of tuples in a relation to the counts of tuples matching a predicate in that relation. We expect it to be slow as it needs to scan through data.

Changelog for-all This implementation does the same as the base for-all operator but uses the changelog table to incrementally compute the current counts. As the changelog table is much smaller in size, it is assumed to be much faster.

4.8 Benchmark results

In the following we present our benchmark results and compare them to the above assumptions. We test selection, join and for-all on live data (*live processing*) and selection and join on past versions (*past processing*). Base approaches shown below set the baseline for each operator in each category.

The scatter plots show the version on the x-axis and the average time to compute completeness on the y-axis in milliseconds.

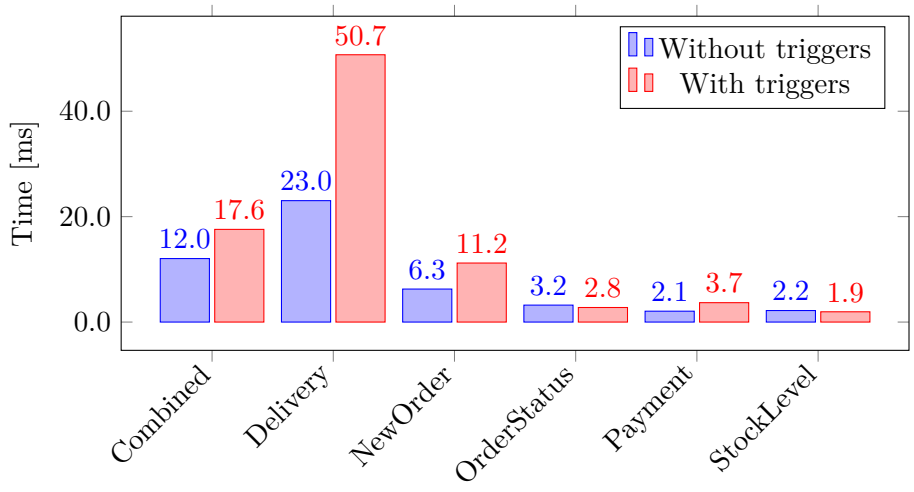

Figure 4.1: Transaction performance with changelog/temporal features and without

4.8.1 Cost of changelog/temporal tables

We measured the cost of maintaining the changelog and temporal tables on Postgres with a trigger-based implementation. The workload is a single-thread TPC-C update process. We compared the results of our benchmark with changelog and temporal features enabled or not. The results have been obtained from the same machine with exactly the same settings otherwise. They are based on running 5000 individual TPC-C transactions. Figure 4.1 compares the performance of running TPC-C with or without additional features. The *combined* entry shows the average time it takes to run a TPC-C transaction, the other entries list the average time a specific type of transaction takes. While order status, payment and stock level transactions perform equally fast, transactions involving data updates are significantly slower. A delivery transaction takes more than twice as long, and a new order nearly twice as long.

To summarize, the performance of writing transactions is cut by half while reading is not affected by the added functionality.

4.8.2 Live processing selection operators

In our benchmark we compare several approaches of selection completeness predicates to each other, as shown in figures 4.2 and 4.3. Note that the selection is applied to a column with a dedicated index, which is also present on the changelog and temporal table.

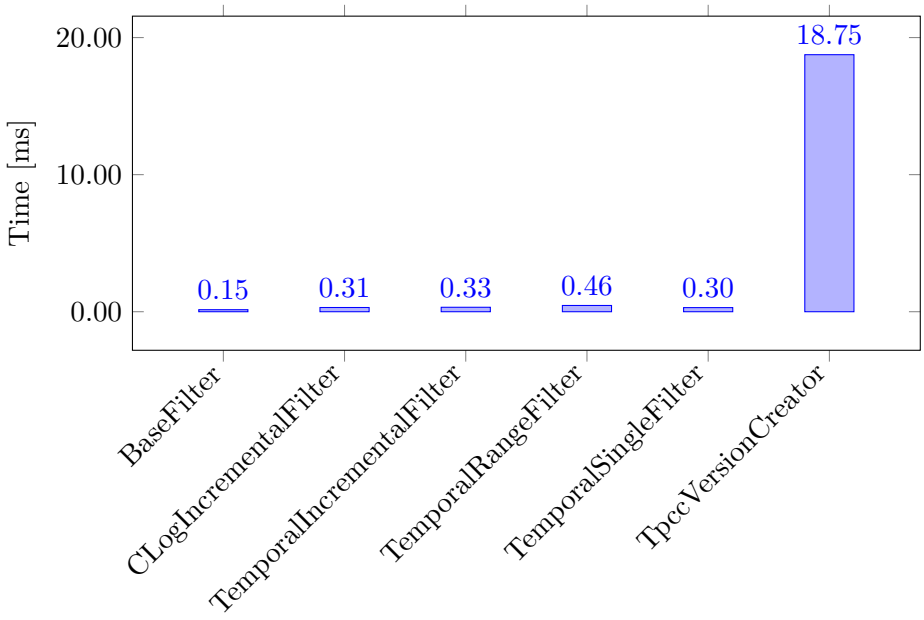

Figure 4.2: Performance of live selection

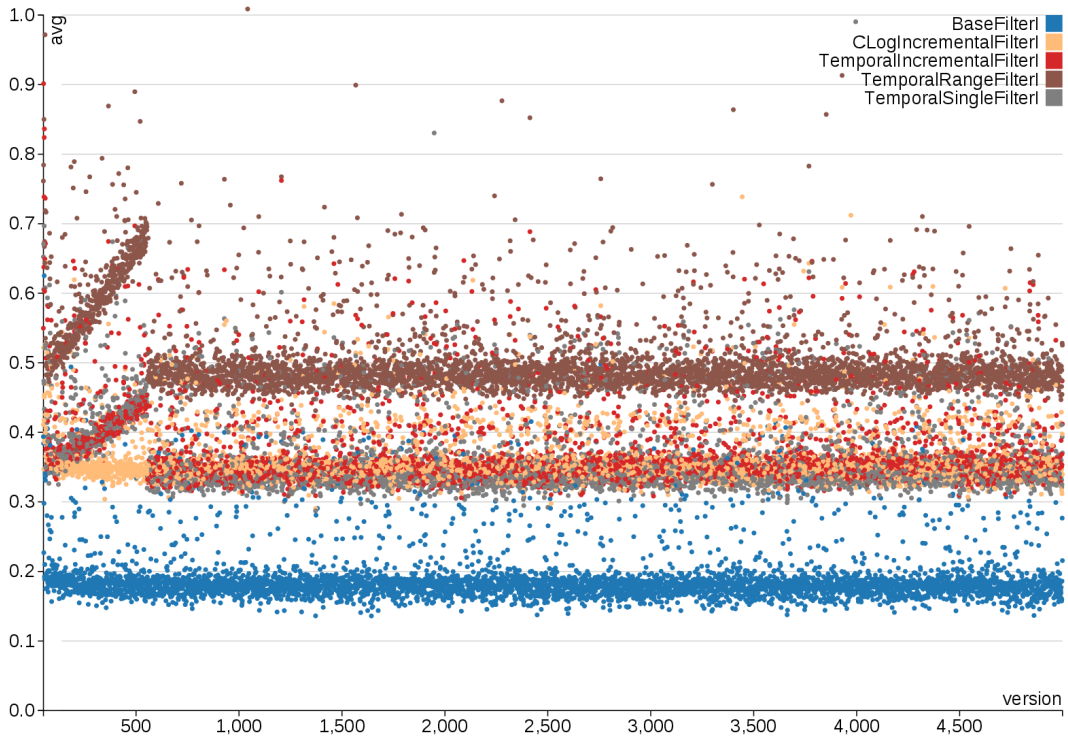

Figure 4.3: Live selection, version number versus average time in milliseconds.

Base selection *(BaseFilterI)*[2] In the incremental case, the base selection is the fastest approach to check a simple selection for completeness, because it is only an index lookup and the SQL expression is very simple. This is the baseline for live selection operators.

Changelog incremental *(CLogIncrementalFilterI)* Testing the changelog for completeness is slower than using the base approach by running the selection on the base table itself. The reason for this is simple: It counts tuples that match the selection criteria, and counts both deletes and inserts. Due to this, the statement is more complex than the base statement.

Temporal incremental *(TemporalIncrementalFilterI)* The temporal incremental approach is about as fast as the changelog approach. This is due to the fact that the query is about as complex as the one for the changelog table but possibly involves more data.

Temporal single version *(TemporalSingleFilterI)* The temporal single version approach (using the `AS-OF` style query) is faster than the range approach, but slower than the base approach. This is expected, as its query is less complex compared to the range approach, but compared to the base approach, the query is more complex, and there is potentially more data involved.

Temporal version range *(TemporalRangeFilterI)* The temporal version range check is the slowest, because it has a complex SQL expression and retrieves a lot of data from the database.

All approaches are very fast, especially the base approach has a negligible cost if the selection is based on an index lookup. The cost for the base approach is close to the minimum of any SQL command, which consists of network communication, command parsing and returning the results.

4.8.3 Live processing join operators

Joins in general are expensive operations because data of two or more sources needs to be combined and matched. A join cannot be efficiently computed without proper indices and advanced algorithms even for small datasets. In this benchmark, we test the performance of an inner join. The results are visualized in figures 4.4 and 4.5.

Base join *(BaseJoinI)* The base join operator executes the join as specified by the user. For processing live data it is the baseline for comparison. In our results it does perform well, but in figure 4.4 it can be seen that it gets slower when the size of the database increases.

[2] Names in parentheses reflect names in the scatter plot.

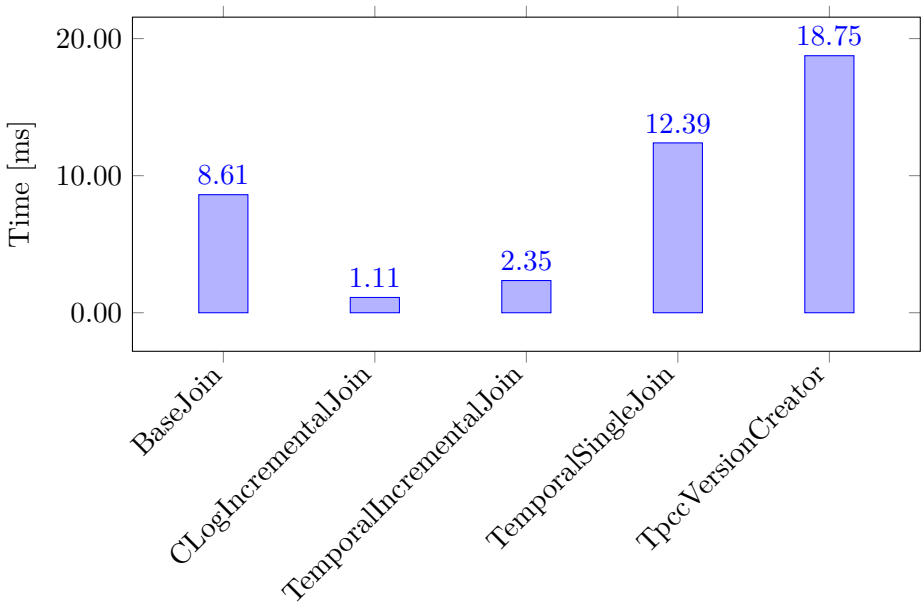

Figure 4.4: Performance of live joins

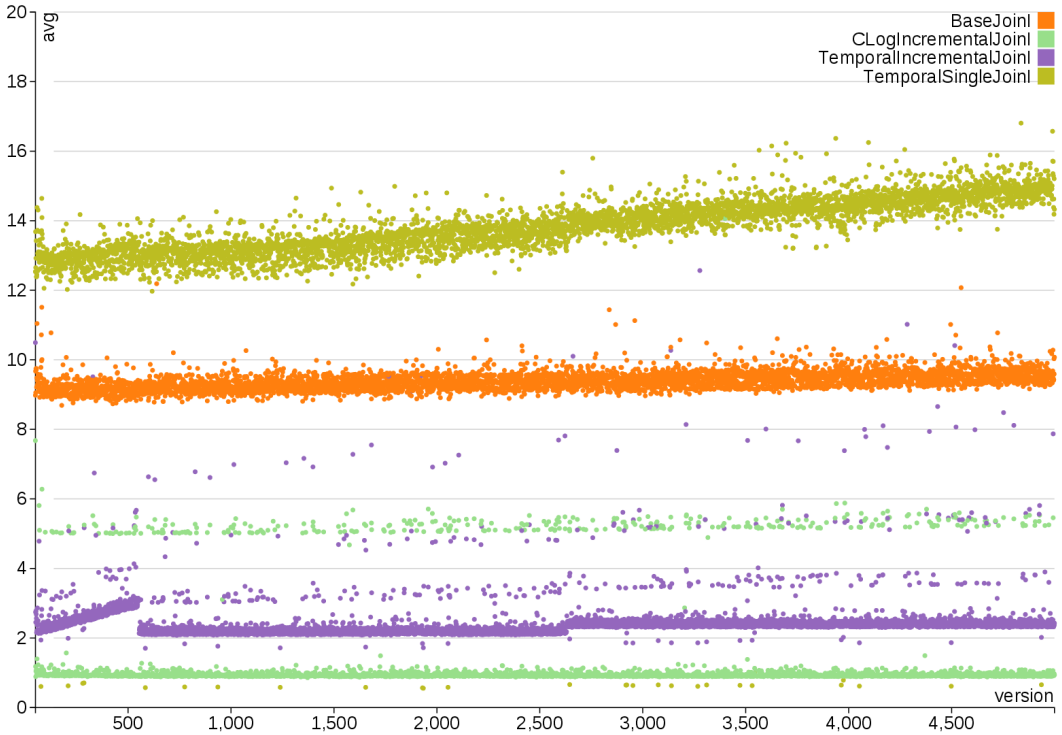

Figure 4.5: Live join, version number versus average time in milliseconds.

Changelog incremental join *(CLogIncrementalJoinI)* Incrementally joining the changelog is the fastest approach in our test. It also shows an interesting pattern consisting of two 'levels' in the scatter plot in figure 4.4. This is due to the fact that the database realizes when there are no changes and the changelog is empty. In this case, it does not perform any operation.

Temporal incremental join *(TemporalIncrementalJoinI)* The temporal incremental join performs slightly worse than the changelog incremental join, due to the fact it has a rather complex SQL expression and the table it operates on potentially contains a lot of data. Still, performance is significantly better than with the base approach because the actual join is computed with less data.

Temporal single version join *CTemporalSingleJoinI)* Recomputing the whole join based on as-of-style queries using the temporal table is slower as the base join (as was to be expected). It gets slower over time as more data is in the temporal table it is working on.

Temporal range join[3] The temporal range join does not perform well. This is mainly related to the Postgres optimizer: It cannot create a good plan for the complex join condition shown in equation 3.50 on page 40. In our tests, it took more than 1000ms to compute completeness on a range spanning one version.

4.8.4 Live processing for-all operators

The result of evaluating different for-all operator implementations is shown in figures 4.6 and 4.7. Note that the for-all operator is based on a selection of a column with index, but as the index does not contain the cardinality of tuples matching a certain condition, it can only be used to extract those tuples. Hence, we observe the following results:

Base for-all *(BaseForAllI)* The base approach simply counts the number of matching tuples versus the number of matching tuples that satisfy the predicate. It is relatively slow as the set of tuples needs to be computed explicitly by the database in order to determine the count.

Changelog for-all *(CLogIncrementalForAllI)* The changelog-based algorithm is significantly faster than the base for-all algorithm. This is due to the small number of tuples that need to be considered when incrementally updating the completeness state.

The result of this evaluation is that the changelog for-all approach is significantly better than the base for-all approach – it even outweighs the costs of maintaining the changelog table.

[3] not shown in scatter plot

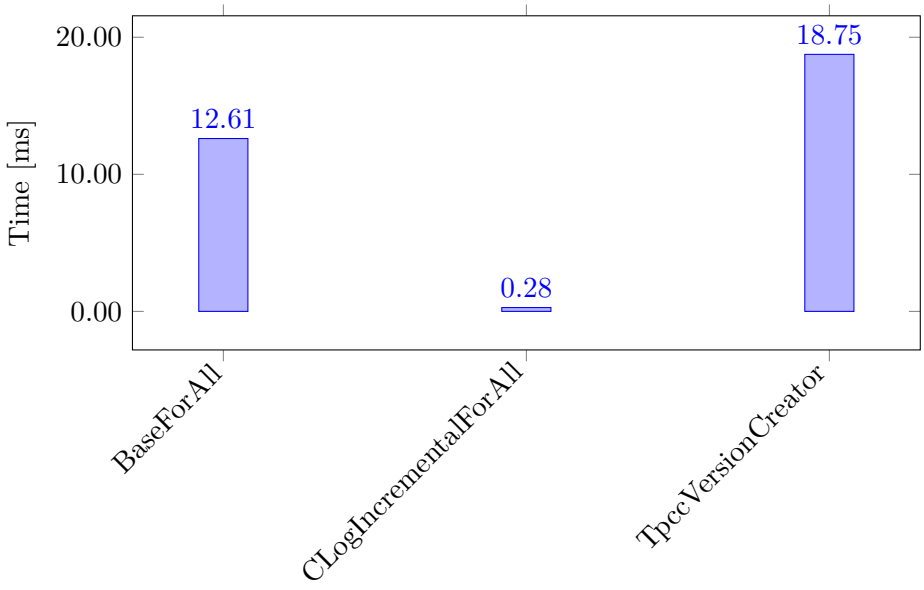

Figure 4.6: Performance of live for-all

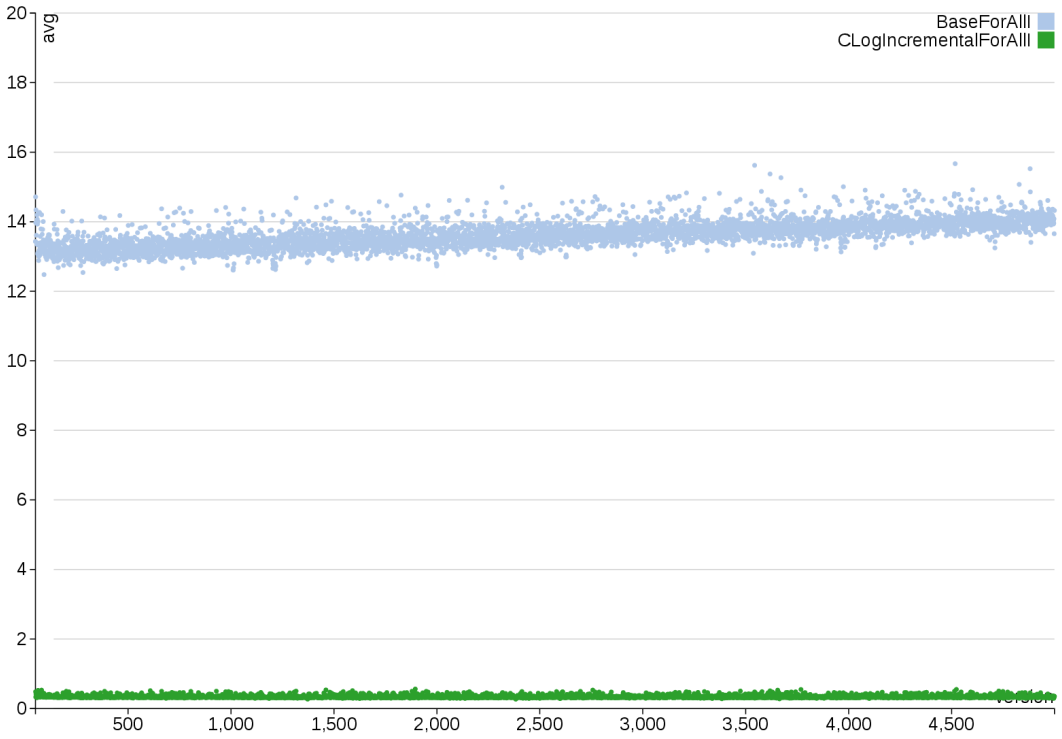

Figure 4.7: Live for-all, version number versus average time in milliseconds.

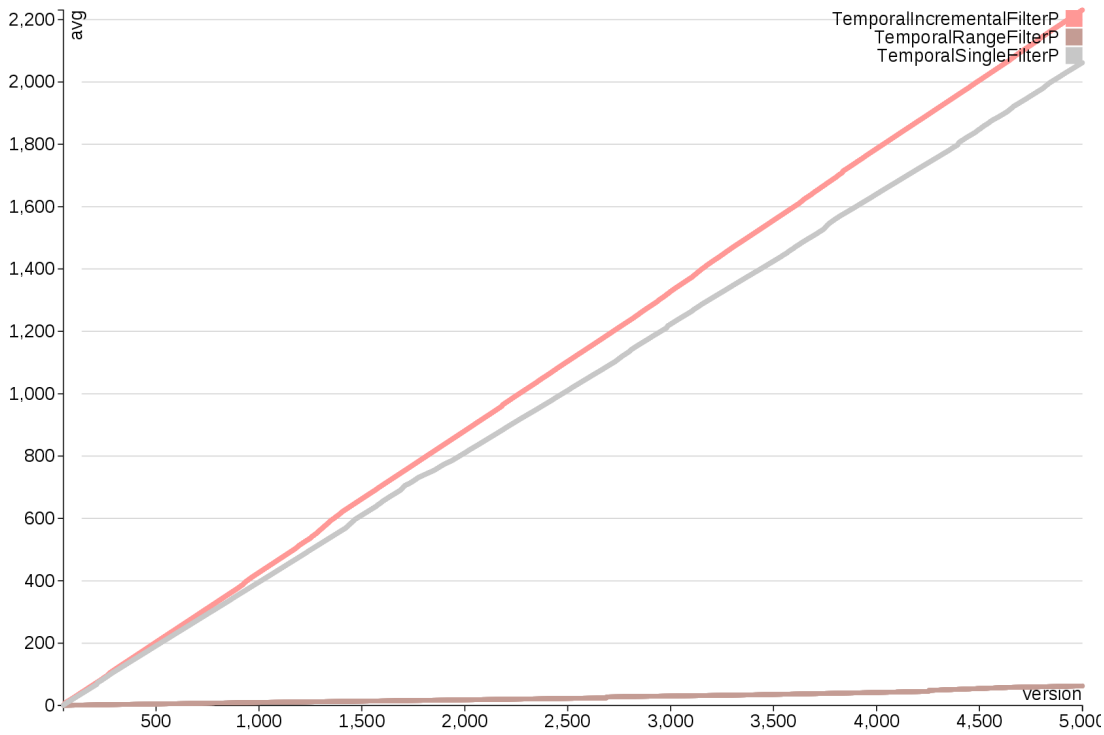

Figure 4.8: Past selection operators, version number versus average time in milliseconds.

4.8.5 Past processing selection operators

To compute completeness over past states using a selection operator yields interesting results (figure 4.8).

Temporal incremental selection *(TemporalIncrementalFilterP)* The incremental selection operation is relatively slow as it needs to communicate with the database on each version. It is the slowest variant in our test.

Temporal single selection *(TemporalSingleFilterP)* The temporal single selection operation also is afflicted by the communication problem the temporal incremental selection suffers from. It is slightly faster than the incremental approach, though.

Temporal range selection *(TemporalRangeFilterP)* The temporal range selection only selects data in a single operation. This causes a slightly increased initial cost, but the cost for checking completeness on following versions is negligible. Thus, it outperforms the other two approaches significantly.

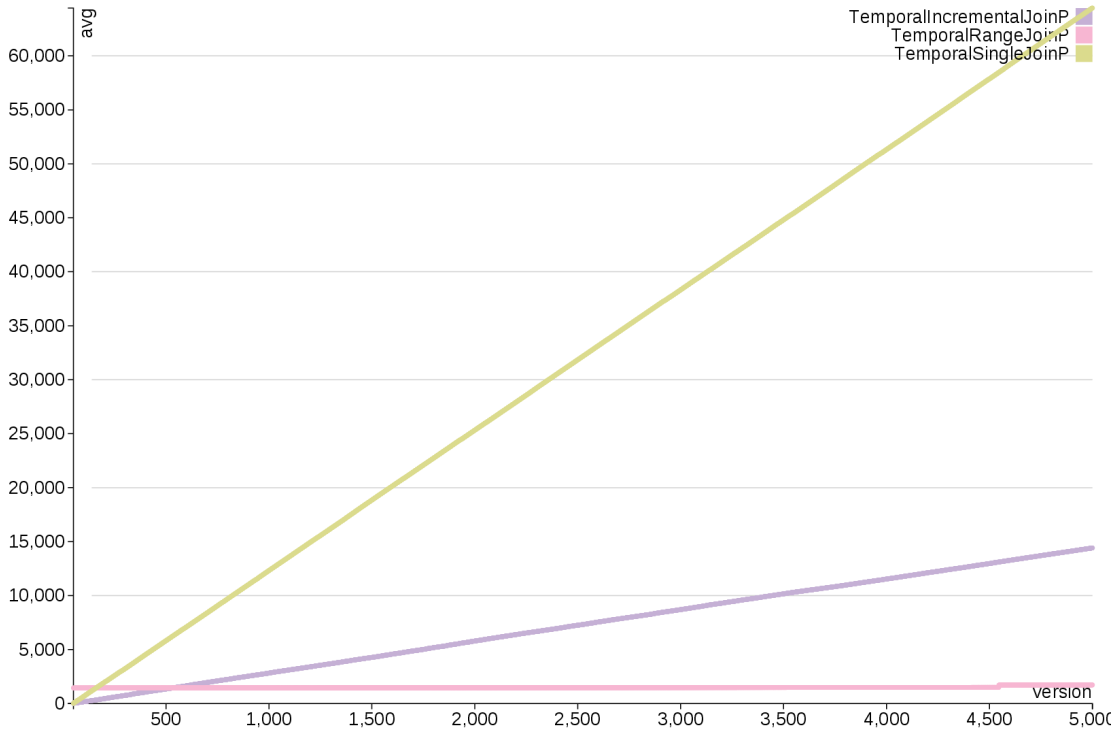

Figure 4.9: Past join, version number versus average time in milliseconds.

4.8.6 Past processing join operators

The past join operator also shows clear results (figure 4.9).

Temporal incremental join *(TemporalIncrementalJoinP)* The temporal incremental join has to select data from the database for every version it is checking for completeness. It is not fast, but as expected, performs better than the temporal single join operator. This is because the database has to retrieve less data for each version.

Temporal single join *(TemporalSingleJoinP)* The temporal single join operator is slow as it computes the whole AS-OF style result for each version.

Temporal range join *(TemporalRangeJoinP)* The temporal range join operator is clearly the best for computing complete versions of past database versions. This is due to the fact that the operator can be expressed as a single SQL operation.

4.8.7 Conclusions

The evaluation of different approaches to implement various temporal completeness operators has shown that there are significant differences in the performance of these

operators. As expected, selection is much faster than join, but depending on the setting, joins can be optimized to reduce the cost compared to the base line – it depends on the use case and environment whether this cost is acceptable or not. Looking at the past operators we can see that it might be better to compute completeness at some point after the transaction has completed at points in time when the database load is lower.

5 Summary

The main goal of this work was to propose a data completeness model for applications accessing a combined data repository. It would allow applications to define constraints that specify what data relevant to them would be complete. We introduced the sandbox concept to provide applications an interface resembling a local database under their possession. We proposed an implementation for sandboxes and analyzed how they can be optimized. Last, we compared various optimization techniques to ensure the viability of the sandbox concept.

The main result of our research is that the sandbox is viable for offering applications an environment where they can express their own completeness and data visibility constraints. We also showed that it is possible to define clear semantics based on a mathematical model. Using this model, we were able to optimize certain classes of completeness predicates in order to be able to quantify the costs associated with an implementation of the sandbox concept.

5.1 Future work

While we researched the fundamentals for the sandbox concept and provided a first approach to quantifying the costs of completeness checking and possible implementations, there arose a magnitude of interesting problems to be studied.

5.1.1 In-depth analysis

We provided a first benchmark to test how completeness operators behave. This benchmark needs to be extended to include past for-all completeness operators. Also, using different sandboxes and other use cases would be interesting and could lead to a better understanding of how sandboxes behave with more and more complex data.

5.1.2 Punctuation

Punctuation is a concept where a data producer indicates the completeness of data by supplying special information. This information can be in the form of a marker tuple. Other applications can define sandboxes that are complete once the marker tuple is present, enabling some form of communication. While not explicitly mentioned in the text, we believe that punctuation can be implemented using selection operators.

5.1.3 Efficiently providing data

At the moment, sandboxes do not remember what part of data was visible in some complete database version. To increase performance, a sandbox could remember which tuples were visible inside a sandbox at which database version, using a specialized index. This could prove useful if the same sandbox is queried repeatedly on the same range of versions.

5.1.4 Remembering complete database versions

A sandbox's completeness predicate evaluated on a specific database version always yields the same result. In order to avoid recomputing completeness for database versions, the sandbox could remember which versions are known to be (in)complete and use this information to speed up finding the next or last complete version. A specialized data structure is required that allows to query least greatest and greatest least neighboring versions given a version to start at.

5.1.5 Branching and writes to sandboxes

There is one problem we did not cover: How to write to sandboxes. We only permit writing to sandboxes if the complete version is the current maximum version of data available in the database, because then it is equal to writing to traditional databases. However, once we would allow writes to past versions many different interesting problems appear. One field of study would be if it were possible to automatically reintegrate branches, or how to merge changes from different branches.

Bibliography

[1] Transaction Processing Performance Council. TPC-C Benchmark Revision 5.10.1.

[2] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, December 1996.

[3] Andreas Behrend, Christian Dorau, and Rainer Manthey. Sql triggers reacting on time events: An extension proposal. In Janis Grundspenkis, Tadeusz Morzy, and Gottfried Vossen, editors, *Advances in Databases and Information Systems*, volume 5739 of *Lecture Notes in Computer Science*, pages 179–193. Springer Berlin Heidelberg, 2009.

[4] Michael Böhlen, Johann Gamper, and Christian S. Jensen. Multi-dimensional aggregation for temporal data. In *Proceedings of the 10th International Conference on Advances in Database Technology*, EDBT'06, pages 257–275, Berlin, Heidelberg, 2006. Springer-Verlag.

[5] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[6] Dengfeng Gao, S. Jensen, T. Snodgrass, and D. Soo. Join operations in temporal databases. *The VLDB Journal*, 14(1):2–29, March 2005.

[7] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *SIGMOD Rec.*, 22(2):157–166, June 1993.

[8] Eric N. Hanson and Lloyd X. Noronha. Timer-driven database triggers and alerters: Semantics and a challenge. *SIGMOD Rec.*, 28(4):11–16, December 1999.

[9] ISO. *ISO/IEC 9075-2:2011 Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[10] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: A unified data structure for processing queries on temporal data in SAP HANA. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1173–1184, New York, NY, USA, 2013. ACM.

[11] N. Kline and R.T. Snodgrass. Computing temporal aggregates. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 222–231, Mar 1995.

[12] Alon Y. Levy. Obtaining complete answers from incomplete databases. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 402–412, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[13] Leonid Libkin. Incomplete data: What went wrong, and how to fix it. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '14, pages 1–13, New York, NY, USA, 2014. ACM.

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

[15] Werner Nutt and Simon Razniewski. Completeness of queries over sql databases. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 902–911, New York, NY, USA, 2012. ACM.

[16] Werner Nutt, Simon Razniewski, and Gil Vegliach. Incomplete databases: Missing records and missing values. In *Proceedings of the 17th International Conference on Database Systems for Advanced Applications*, DASFAA'12, pages 298–310, Berlin, Heidelberg, 2012. Springer-Verlag.

[17] Simon Razniewski and Werner Nutt. Completeness of queries over incomplete databases. *PVLDB*, 4(11):749–760, 2011.

[18] Betty Salzberg and Vassilis J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, June 1999.

[19] Ognjen Savković, Mirza Paramita, Sergey Paramonov, and Werner Nutt. Magik: Managing completeness of data. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 2725–2727, New York, NY, USA, 2012. ACM.

[20] Richard T Snodgrass. *The TSQL2 temporal query language*, volume 330. Springer, 1995.

[21] Andrew Witkowski, Srikanth Bellamkonda, Hua-Gang Li, Vince Liang, Lei Sheng, Wayne Smith, Sankar Subramanian, James Terry, and Tsae-Feng Yu. Continuous queries in oracle. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1173–1184. VLDB Endowment, 2007.

[22] Donghui Zhang, V.J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 103–113, 2002.