CDMTCS
Research
Report
Series

Index Design for Partial Referential Integrity

Mozhgan Memari
University of Auckland,
Auckland, New Zealand

Sebastian Link
University of Auckland,
Auckland, New Zealand

CDMTCS-482
March 2015

Centre for Discrete Mathematics and
Theoretical Computer Science

Index Design for Partial Referential Integrity

Mozhgan Memari
University of Auckland, New Zealand
m.memari@auckland.ac.nz

Sebastian Link
The University of Auckland, New Zealand
s.link@auckland.ac.nz

March 18, 2015

Abstract

Referential integrity is fundamental for data processing and data quality. The SQL standard proposes different semantics under which referential integrity can be enforced in practice. Under simple semantics, only total foreign key values must be matched by some referenced key values. Under partial semantics, total and partial foreign key values must be matched by some referenced key values. Support for simple semantics is extensive and widespread across different database management systems but, surprisingly, partial semantics does not enjoy any native support in any known systems. Previous research has left open the questions whether partial referential integrity is useful for any real-world applications and whether it can enjoy efficient support at the systems level. As our first contribution we show that efficient support for partial referential integrity can provide database users with intelligent query and update services. Indeed, we regard partial semantics as an effective imputation technique for missing data in query answers and update operations, which increases the quality of these services. As our second contribution we show how partial referential integrity can be enforced efficiently for real-world foreign keys. For that purpose we propose triggers and exploit different index structures. Our experiments with synthetic and benchmark data sets confirm that our index structures do not only boost the performance of the state-of-the-art recommendation for enforcing partial semantics in real-world foreign keys, but show trends that are similar to enforcing simple semantics.

Keywords: Data quality; Incomplete data; Index; Null markers; Performance evaluation; Uncertain data

1 Introduction

In his seminal paper [5] Codd introduced the principles of entity and referential integrity as two fundamental cornerstones of the relational model of data. While more than 100 classes of relational integrity constraints have been investigated [21], relational database management systems offer only native support for keys and foreign keys, which enforce entity and referential integrity, respectively. Indeed, keys and foreign keys provide principled mechanisms to process quality data efficiently. The SQL standard promotes the use of two different semantics for referential integrity. Under *simple* semantics, referencing tuples with null markers on some of their foreign key columns satisfy referential integrity by default. Under *partial* semantics, every referencing tuple requires a referenced tuple that matches all total values on the foreign key columns in the corresponding key columns. Partial semantics offers a higher degree of data quality as it subsumes simple semantics.

Example 1 *For illustration consider an example from an Australian tourism company [8]. Tours in the* TOUR *table have a tour_id, for example a tour such as the "Gold Coast Grand Tour" has tour_id GCG. Tours have a fixed sequence of sites they visit. Sites are identified by a site_code (e.g., MV) but also have a unique site_name (e.g., Movie World). The primary key on* TOUR *is* $\{tour\_id, site\_code\}$. *Booking orders by visitors, who can join a tour from any allocated site, are stored in the* BOOKING *table. The foreign key*

$$[tour\_id, site\_code] \subseteq \text{TOUR}[tour\_id, site\_code]$$

is defined on BOOKING. *Consider the following database.*

TOUR

Tour_id	***Site_code***	***Site_name***
GCG	*OR*	*O'Reilly's*
BRT	*OR*	*O'Reilly's*
BRT	*MV*	*Movie World*
RF	*BB*	*Binna Burra*
RF	*OR*	*O'Reilly's*

BOOKING

Visitor_id	***Tour_id***	***Site_code***	***Date***
1006	*BRF*	`null`	*Sep* 19^{th}
1001	*BRT*	*OR*	*Nov* 21^{st}
1008	`null`	*BB*	*Sep* 5^{th}
1012	`null`	*BR*	*Nov* 2^{nd}
1011	*RF*	`null`	*Oct* 5^{th}

Simple referential integrity is satisfied: the only total foreign key value (BRT,OR) in the BOOKING *table is matched in the* TOUR *table. Partial referential integrity is violated: for the partial foreign key value (BRF,*`null`*) in the* BOOKING *table there is no tuple in the* TOUR *table with value "BRF" on tour_id, and similarly for (*`null`*, BR).*

While every database management system we know offers built-in support to enforce simple referential integrity, it is surprising that none of them offers built-in support to enforce partial referential integrity [22]. Explanations for this gap between the SQL standard and its implementations have been brought forward by Härder and Reinhart, who analyzed in great detail the requirements of partial referential integrity on the operational level [9]. Two main questions remain open two decades after they have been posed in [9]:

1. Is partial referential integrity useful for any real-world application?
2. Can partial referential integrity be enforced efficiently?

Contribution. In the present paper we provide affirmative answers to both questions. Our first main contribution shows that partial referential integrity is useful for the two most significant real-world applications of database technology: updates and queries. More specifically, we propose intelligent update and query services that are based on efficiently enforcing partial semantics. Indeed, partial referential integrity can be exploited to impute missing data values. Our intelligent updates provide database users with sound choices to reduce the level of incompleteness in the database, which is an important measure for the quality of data [7]. In the example above, suppose that the last tuple is not already part of the Booking table but about to be inserted. Based on the assumption that the updated table shall satisfy partial referential integrity, our update service would provide the user with the option to replace the null marker by either "BB" or "OR". This service can be customized further depending on the authorization rights of the user, for example. Our intelligent query service provides database users with additional query answers that result from the imputation of missing data values in standard answers. In our example, the standard answer to the query that selects *tour_id* and *site_code* from tuples in the Booking table, may be augmented by the tuples (RF,BB), and (RF,OR) based on the partial semantics of the foreign key. We believe that intelligent updates and queries offer a strong affirmative answer to the first open question, that goes beyond the straightforward argument that partial referential integrity targets a higher level of data quality than simple referential integrity. Note that our applications cannot be supported by simple semantics. Indeed, they provide strong motivation to investigate the second open question, as the effectiveness of the applications largely depend on the efficiency of enforcing partial semantics for real-world foreign keys. Based on our experience, the literature [6] and public schemata, most real-world foreign keys have rarely more than four columns and the referenced key is commonly the primary key, or a candidate key where all columns are `NOT NULL`. Our second main contribution is a detailed analysis of partial referential integrity at the systems level, as proposed for future work in [9]. The main finding is that partial semantics for "real-world" foreign keys can be implemented in the form of triggers and enforced efficiently by a right combination of indices. In general, it is worthwhile investigating which subsets of the foreign key columns carry the most total values, and then define indices on those subsets. As updates include the maintenance of all affected indices, having too many indices means that the loss in time for their maintenance outweighs the gain in search time when enforcing referential integrity. Some organizations may have a good knowledge of the top-k indices they want to support, but we have made it part of our research to shed further light on what a reasonable number k could be. For this purpose, we applied our analysis to test data in which null marker occurrences are evenly distributed between all possible subsets of columns. That is, we have the least degree of information available about which indices to define. Our recommendation for an n-column foreign key is to exploit $n+1$ indices on each of the referencing and referenced tables: one compound index on the n columns, and one index on each of the n individual columns. This combination of indices outperforms any other combination for all possible kinds of updates. The finding is confirmed by experiments on two benchmark and one real life database. We remark that the original proposal by Härder and Reinhart to utilize one index for each of the n key columns on the referenced table and one compound index on the foreign key columns of the referencing table performs well only for 2-column key relationships. Essentially, by doubling the number of indices over their proposal, we improve the speed for inserts by a factor of 7, and for deletions by a factor of 123, on the largest data set considered with a 5-column foreign key relationship. Note that this includes the maintenance of all indices involved. Further experiments explain this performance boost over the original proposal: Adding an index for each of the n foreign key columns on the referencing table overcomes the poor performance of the original proposal when deleting tuples from the referenced table that have no alternative match for referencing tuples. Furthermore, adding a single index for the compound key on the referenced table boosts the performance when inserting total tuples in the referencing table. The only trade-off we found is that the time for loading data on the referencing table is 1.5 times more, due to building twice as many indices. This one-time cost is feasible.

Organization. The remainder of this paper is organized as follows. We comment on some related work in Section 2. Details on the semantics of referential integrity constraints in SQL are given in Section 3. We describe our ideas of intelligent update and query services in Sections 4 and 5, based on partial semantics. In Section 6 we propose triggers as well as five different index structures for which we will analyze the performance of enforcing partial referential integrity on synthetic data sets in Section 7, and on TPC-C and TPC-H data sets in Section 8. We conclude in Section 9 where we also briefly comment on future work.

2 Related Work

Work on referential integrity has largely addressed inclusion dependencies. A seminal paper on the theory of inclusion dependencies is [2], in which a finite axiomatization for the associated implication problem is presented and the non-k-ary-axiomatizability of both finite and unrestricted implication for functional and inclusion dependencies together is demonstrated. An axiomatization which is not k-ary for the finite implication of functional and inclusion dependencies is presented in [18]. Undecidability of (finite) implication for functional and inclusion dependencies taken together was shown independently by [4] and [19]. Another seminal paper is [12], which also observed the distinction between finite and unrestricted implication for functional and inclusion dependencies, generalized the chase to incorporate functional and inclusion dependencies, and used this to characterize containment between conjunctive queries. Databases have benefited from referential integrity constraints and inclusion dependencies in areas as diverse as database design [15], consistency enforcement [3], query optimization [12], data cleaning [1], data quality [20], and data profiling [24]. Inclusion dependencies have also been considered in XML [13] and RDF [14].

The different semantics of referential integrity, as proposed by the SQL standard [17], have not received much attention from neither academia nor practice. As observed in [22], there are no database management systems that offer built-in support for enforcing partial referential integrity while every database management system offers built-in support for enforcing simple referential integrity. In [9] Härder and Reinhart investigated the functional requirements for preserving simple and partial referential integrity. Indeed, they determined the number and kinds of searches necessary for referential integrity maintenance, without implementation considerations. Their main result was that a combined access path structure is the most appropriate for checking simple semantics, while partial semantics requires very expensive and complicated check procedures. Their "best advice is to avoid the use of `MATCH PARTIAL` at all". If required, they recommend the use of one index for each of the key columns on the referenced table and one compound index on the foreign key columns of the referencing table. They also investigate the performance of multi-dimensional access paths by considering grid file structures. Here, the access costs for partial match queries are remarkably more expensive than their suggested index option. The main reason is that grid files retrieve all matching tuples while partial referential integrity requires only one matching tuple. In conclusion, Härder and Reinhart say that their "presented results rely on the assumption that the search costs are indicative for the entire costs of referential integrity maintenance. This assumption has to be justified through further research especially at the system level. Another interesting question to be answered is whether or not `MATCH PARTIAL` is useful for a real world application". Here, we address both questions.

In a research-in-progress paper [16] we performed a static analysis of the costs for validating simple and partial semantics in a fixed database. Therefore, the analysis did not consider updates at all. It only applied to referential integrity constraints with two columns, and only considered compound indices which refer to all columns of a foreign key. The present paper presents a detailed analysis of the costs for enforcing simple and partial semantics in a dynamic database that is subject to updates; applies to foreign keys with up to five columns, considers multiple index structures on referenced and referencing tables, and proposes triggers and the new intelligent query and update services.

3 Referential Integrity in SQL

Foreign keys form one of the most fundamental classes of integrity constraints, which implement Codd's proposal of referential integrity from his seminal paper [5]. Referential integrity maintains the relationship between two table schemata, which are the referencing schema or child schema, usually denoted by C_S, and the referenced schema or parent schema, usually denoted by P_S. A referential integrity constraint is commonly written as

$$[f_1, \dots, f_n] \subseteq P_S[k_1, \dots, k_n]$$

to denote the relationship between the sequence $[f_1, f_2, \ldots, f_n]$ of distinct column names on C_S, usually called the foreign key, and the sequence $[k_1, k_2, \ldots, k_n]$ of distinct column names, which form a candidate key on P_S. For $i = 1, \ldots, n$, the domains of the column names f_i and k_i must match. Intuitively, referential integrity requires that for each tuple c in a child table C there is a matching tuple p in the parent table P. The SQL standard recommends the use of the `MATCH` clause to specify different ways for handling occurrences of the null marker `null` in foreign key and key columns [17].

Under *simple* semantics, the foreign key constraint is satisfied if for every tuple c in the child table C, either $c(f_i) = \texttt{null}$ for some $1 \leq i \leq n$, or there is some tuple p in the parent table P such that $c(f_i) = p(k_i)$ for all $i = 1, \ldots, n$. More precisely,

$$\forall c \in C \left(\left(\bigwedge_{i=1}^{n} c(f_i) \neq \texttt{null} \right) \Rightarrow \exists p \in P \left(\bigwedge_{i=1}^{n} c(f_i) = p(k_i) \right) \right) .$$

Hence, simple referential integrity is never violated by tuples that are partially defined on the foreign key columns.

Under *partial* semantics, the foreign key constraint is satisfied if for every tuple c in the child table C there is some tuple p in the parent table P such that $c[f_1, \ldots, f_n]$ is subsumed by $p[k_1, \ldots, k_n]$. That is,

$$\forall c \in C \exists p \in P \left(c[f_1, \ldots, f_n] \sqsubseteq p[k_1, \ldots, k_n] \right)$$

Here, a tuple c over the column sequence $[f_1, \ldots, f_n]$ is *subsumed* by a tuple p over the column sequence $[k_1, \ldots, k_n]$, if for every $i = 1, \ldots, n$, $c(f_i) = \texttt{null}$ or $c(f_i) = p(k_i)$.

The table from Example 1 satisfies the foreign key constraint $[\mathit{tour\_id}, \mathit{site\_code}] \subseteq$ TOUR $[\mathit{tour\_id}, \mathit{site\_code}]$ on table BOOKING under simple semantics, but not under partial semantics. For instance, the BOOKING-tuple (BRF,`null`) over $[\mathit{tour\_id}, \mathit{site\_code}]$ has no TOUR-tuple over $[\mathit{tour\_id}, \mathit{site\_code}]$ by which it is subsumed.

Enforcing some referential integrity constraint means that each time the child or parent table is modified it must be verified that the constraint is satisfied by the modified database instance. Therefore, referential integrity is particularly important for transactional databases, or for updates of important data such as master data.

In general, six basic updates must be addressed to accommodate all possible modifications on parent tables P or child tables C. Tuple inserts into P and tuple deletions from C do not cause a violation of referential integrity. The other four update operations, however, ⟨*Insert a new tuple into C*⟩ and ⟨*Update C*⟩, ⟨*Update P*⟩ and ⟨*Delete a tuple from P*⟩ may lead to modified tables that violate referential integrity. If a tuple c from C references a tuple p from P, we call c a child of p, and p a parent of c.

⟨*Delete a tuple from P*⟩: A tuple p from P may be the only parent of some child c from C. That is, there is no other tuple p' in P which is a parent of c. Deleting such single parents from P will always violate referential integrity.

⟨*Update P*⟩: This update can be interpreted as ⟨*Delete a tuple from P*⟩ along with ⟨*Insert a new tuple into P*⟩. Therefore, it may only cause a referential integrity violation due to the Delete action.

⟨*Insert a new tuple into C*⟩: Referential integrity requires for every newly inserted child c in C the existence of a parent p in P. Otherwise, referential integrity is violated.

$\langle$*Update C*$\rangle$: An update of a child in C can be interpreted as a delete from C followed by an insert into C. Only the insert into C can lead to a violation of referential integrity.

According to the SQL standard, inserts into C or updates on C are only allowed if they result in a new child table that satisfies the referential integrity constraints defined on C_S. However, different actions can be applied when a delete from P or an update on P results in the violation of some referential integrity constraint. Based on the SQL standard, "CASCADE", "SET NULL", "SET DEFAULT", "RESTRICT" and "NO ACTION" are available referential actions.

Under simple semantics, every child has at most one parent. Under partial semantics, any child may have several parents, given that the child has a null marker occurrence in some of its foreign key columns. If the state of a child is defined as the subset of the n foreign key columns on which it is `null`, then each given parent may have up to $2^n - 1$ children that have pairwise different states. If u denotes the number of null marker occurrences $(0 \leq u < n)$, $\binom{n}{u}$ is the number of distinct states with u null marker occurrences [9].

Example 2 *Given a 3-attribute key with value* $\langle 1,2,3\rangle$ *on the key columns, the seven different states may result from children with the following values on their foreign key columns:* $\langle 1,2,3\rangle$, $\langle \texttt{null},2,3\rangle$, $\langle 1,\texttt{null},3\rangle$, $\langle 1,2,\texttt{null}\rangle$, $\langle \texttt{null}, \texttt{null},3\rangle$, $\langle \texttt{null},2,\texttt{null}\rangle$ *and* $\langle 1,\texttt{null},\texttt{null}\rangle$. *Each of the non-total children may have some other parent. The parent with* $\langle 4,2,3\rangle$, *for example, has also children with the following values on their foreign key columns* $\langle \texttt{null},2,3\rangle$, $\langle \texttt{null},2,\texttt{null}\rangle$ *and* $\langle \texttt{null},\texttt{null},3\rangle$.

4 An Intelligent Update Service

We propose an intelligent update service that enables us to give an affirmative answer to Härder and Reinhart's question "whether or not `MATCH PARTIAL` is useful for a real world application?" [9].

Null markers offer great flexibility for data entry, but have serious consequences. Indeed, the level of information completeness is an important factor for data quality [7]. Partial data causes significant problems for enterprises: it routinely leads to misleading analytical results and biased decisions, and accounts for loss of revenue, credibility and customers [7]. We propose the use of `MATCH PARTIAL` for intelligent updates of data. For a given foreign key $[f_1, \ldots, f_n] \subseteq P_S[k_1, \ldots, k_n]$ on C_S we propose the following two strategies to reduce information incompleteness.

4.1 Intelligent Insertions

Suppose a new tuple c is inserted into the child table C and $c(f_i) = \texttt{null}$ on some foreign key column $f_1, \ldots, f_n$. Then the database management system (DBMS) determines all parents p of the parent table P where $c[f_1, \ldots, f_n] \sqsubseteq p[k_1, \ldots, k_n]$. For each of these parents p, the DBMS computes the tuple c_p that results from c by replacing each null marker occurrence $c(f_i) = \texttt{null}$ by the value $p(k_i)$, where $i = 1, \ldots, n$. Finally, the tuples c_p are presented as alternatives to c for insertion into C.

For instance, suppose again that the tuple $c = (1011, \text{RF}, \texttt{null}, \text{Oct } 5^{th})$ is not already part of the BOOKING table in Example 1, but about to be inserted into it. Our intelligent update service would determine all possible parents of c in TOUR, which are $p = (\text{RF}, \text{BB}, \text{Binna Burra})$ and $p' = (\text{RF, OR, O'Reilly's})$, and present the tuples $c_p = (1011, \text{RF}, \text{BB}, \text{Oct } 5^{th})$ and $c_{p'} = (1011, \text{RF}, \text{OR}, \text{Oct } 5^{th})$ as alternatives to $c = (1011, \text{RF}, \texttt{null}, \text{Oct } 5^{th})$ for insertion into BOOKING.

4.2 Intelligent Deletions

Suppose an existing tuple p is deleted from P. For all children c of p in C where $c[f_1, \ldots, f_n] \sqsubseteq p[k_1, \ldots, k_n]$ and $c(f_i) = \texttt{null}$ on some $f_1, \ldots, f_n$, the DBMS determines all alternative parents of p in $P - \{p\}$, i.e. those tuples $p' \in P - \{p\}$ where $c[f_1, \ldots, f_n] \sqsubseteq p'[k_1, \ldots, k_n]$. It then computes the tuple $c_{p'}$ that results from c by replacing each null marker occurrence $c(f_i) = \texttt{null}$ by the value $p'(k_i)$, where $i = 1, \ldots, n$. Finally, for each child c of p the tuples $c_{p'}$ are presented as potential updates of c in C. For instance, consider the tables from Example 1. Suppose the tuple $p = (\text{RF}, \text{OR}, \text{O'Reilly's})$ is deleted from the TOUR table. The only child of p in BOOKING is $c = (1011, \text{RF}, \texttt{null}, \text{Oct } 5^{th})$, and the only alternative parent of c in TOUR is $p' = (\text{RF}, \text{BB}, \text{Binna Burra})$. Consequently, the DBMS presents the tuple $c_{p'} = (1011, \text{RF}, \text{BB}, \text{Oct } 5^{th})$ as an update of the tuple c in BOOKING.

We propose two different approaches for implementing intelligent deletions. In Method 1, the existence of alternative parents is first checked in P. Then the potential updates are ranked according to the number of affected children in C and presented to the user for confirmation.

In Method 2, the intelligent system first finds all children of the given parent. For each of these children, the system then finds all alternatives. The choice of method depends on the requirements of the application.

4.3 Implementation

The system is available on `http://sqlkeys.info` and has been tested on the 3-column foreign key of the TPC-C benchmark database from the Transaction Processing Performance Council (`http://www.tpc.org`). Figure 1 shows how users can insert either their original record or a more informative record, whatever is perceived to be the better choice. Figure 2 shows screenshots of the two deletion methods. Indeed, the choice of continuing with incomplete foreign keys is available to users, however, referential action will be applied on the foreign keys which violate partial referential integrity.

Use cases. It is straightforward to envision novel use cases of the intelligent update service. For example, when updates are run manually, the user can be presented directly with available choices for the imputation of null markers. This can be customized further, for example, according to the preferred number of such choices or to the access rights the user enjoys. The decision should be based on the efficiency and quality requirements for data entry as well as the expertise of the user. When updates are run mechanically, it is particularly advisable to record the available choices for imputation in the form of a log. This log can be inspected later on for analytical purposes, or to assist with data

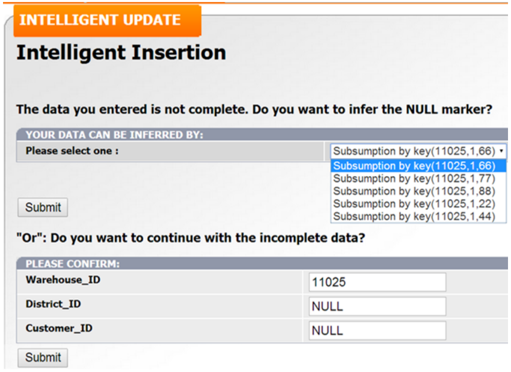
Algorithm 1 Intelligent Deletion: Method 1

```
Require: Deleted tuple: $p[k_1, \ldots, k_n]$, referential action
Ensure: Updated $C$ under Partial Semantics
for all $c \in \mathbf{C}$ such that $\bigwedge_{i=1}^{n} c(f_i) \neq$ null and $\bigwedge_{i=1}^{n} c(f_i) = p(k_i)$ do
    Apply referential action
for $u = 1 \rightarrow n-1$ do
    $S[u] \leftarrow \{S_{uj} |$ the $j^{th}$ state of $\langle k_1, \ldots, k_n \rangle$ with $u$ nulls, for all j=1 to $\binom{n}{u}\}$
    for all $S_{uj} \in S[u]$ do
        $Q[S_{uj}] \leftarrow \{p' \in P - \{p\} | S_{uj} \sqsubseteq p'[k_1, \ldots, k_n]\}$
        $l_{uj} \leftarrow$ number of $c$ in $C$ match $S_{uj}$
        $l_{mj} \leftarrow$ number of $c$ in $C$ match $S_{mj}$ such that $m = u+1$ to $n-1$ and $S_{mj} \sqsubseteq S_{uj}$
        $l'_{uj} \leftarrow l_{uj} + l_{mj}$
        if $Q[S_{uj}] = \emptyset$ and $l_{uj} \neq 0$ then
            Apply referential action on $S_{uj}$ states in $C$
            $l'_{uj} \leftarrow 0$
    $L \leftarrow \{(l'_{uj}, Q[S_{uj}]\}$
    if $\exists\ l'_{uj} \neq 0 \in L$ then
        $S'_u \leftarrow S_{uj}$ with Max$(l'_{uj})$
        Output: Set $Q[S'_u]$ and $S'_u$ (foreign key)
        Input: If $p' \in Q[S'_u]$ is selected then subsume all $c = S_{uj}$ and $c = S_{mj}$ by $p'$
        $l'_{uj} \leftarrow 0$ and $L \leftarrow \{(l'_{uj}, Q[S'_u]\}$
```

Algorithm 2 Intelligent Deletion: Method 2

```
Require: Deleted tuple: $p[k_1, \ldots, k_n]$, referential action
Ensure: Updated $C$ under Partial Semantics
for all $c \in \mathbf{C}$ such that $\bigwedge_{i=1}^{n} c(f_i) \neq$ null and $\bigwedge_{i=1}^{n} c(f_i) = p(k_i)$ do
    Apply referential action
for $u = 1 \rightarrow n-1$ do
    $S[u] \leftarrow \{S_{uj} |$ the $j^{th}$ state of $\langle k_1, \ldots, k_n \rangle$ with $u$ nulls, for all j=1 to $\binom{n}{u}\}$
    for all $S_{uj} \in S[u]$ do
        $l_{uj} \leftarrow$ number of $c$ in $C$ match $S_{uj}$
    $L \leftarrow \{(l_{uj}, Q[S_{uj}]\}$
    if $\exists\ l_{uj} \neq 0 \in L$ then
        $S'_u \leftarrow S_{uj}$ with Max$(l_{uj})$
        $Q[S'_u] \leftarrow \{p' \in P - \{p\} | S'_u \sqsubseteq p'[k_1, \ldots, k_n]\}$
        if $Q[S'_u] = \emptyset$ then
            Apply referential action on $S'_u$ states in $C$
        else
            Output: Set $Q[S'_u]$ and $S'_u$ (foreign key)
            Input: If $p' \in Q[S'_u]$ is selected then subsume all $c = S_{uj}$ and $c = S_{mj}$ by
            $p'$ where $m = u+1$ to $n-1$ and $S_{mj} \sqsubseteq S_{uj}$
            $l_{uj} \leftarrow 0$ and $L \leftarrow \{(l_{uj}, Q[S'_u]\}$
```

Figure 1: Intelligent Update System: Insertion



Intelligent Insertion

The data you entered is not complete. Do you want to infer the NULL marker?

YOUR DATA CAN BE INFERRED BY:

Please select one : Subsumption by key(11025,1,66)

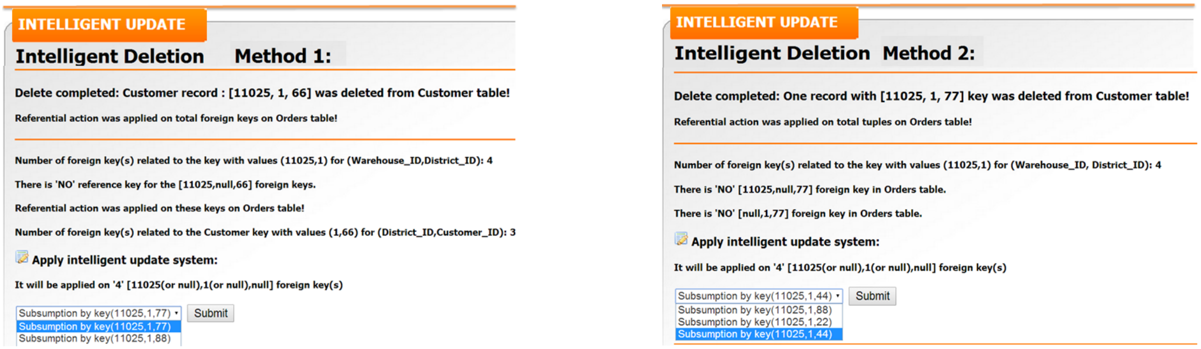
Subsumption by key(11025,1,66)
Subsumption by key(11025,1,77)
Subsumption by key(11025,1,88)
Subsumption by key(11025,1,22)
Subsumption by key(11025,1,44)
Submit

"Or": Do you want to continue with the incomplete data?

PLEASE CONFIRM:

Warehouse_ID	11025
District_ID	NULL
Customer_ID	NULL

Figure 2: Intelligent Deletion Methods 1 and 2



Intelligent Deletion Method 1:

Delete completed: Customer record : [11025, 1, 66] was deleted from Customer table!

Referential action was applied on total foreign keys on Orders table!

Number of foreign key(s) related to the key with values (11025,1) for (Warehouse_ID,District_ID): 4

There is 'NO' reference key for the [11025,null,66] foreign keys.

Referential action was applied on these keys on Orders table!

Number of foreign key(s) related to the Customer key with values (1,66) for (District_ID,Customer_ID): 3

Apply intelligent update system:

It will be applied on '4' [11025(or null),1(or null),null] foreign key(s)

Subsumption by key(11025,1,77)

Submit
Subsumption by key(11025,1,77)
Subsumption by key(11025,1,88)
INTELLIGENT UPDATE

Intelligent Deletion Method 2:

Delete completed: One record with [11025, 1, 77] key was deleted from Customer table!

Referential action was applied on total tuples on Orders table!

Number of foreign key(s) related to the key with values (11025,1) for (Warehouse_ID, District_ID): 4

There is 'NO' [11025,null,77] foreign key in Orders table.

There is 'NO' [null,1,77] foreign key in Orders table.

Apply intelligent update system:

It will be applied on '4' [11025(or null),1(or null),null] foreign key(s)

Subsumption by key(11025,1,44)

Submit
Subsumption by key(11025,1,88)
Subsumption by key(11025,1,22)
Subsumption by key(11025,1,44)

cleaning. An interesting use case occurs whenever transactions are aborted due to null marker occurrences in child columns that are part of the primary key. Exploiting partial semantics to impute these occurrences by some matching consistent values may result in the successful completion of the transaction. In any case, the intelligent update service can help reduce information incompleteness in ways current services cannot.

5 An Intelligent Query Service

The occurrence of null markers in query answers restricts the insights that stakeholders can gain from data. It is therefore important that database systems raise user awareness of actual data values that null markers may represent. By example, we will now explain why partial referential integrity constitutes a prime mechanism to reduce information incompleteness in query answering. Consider the following query which returns the *tour_id* and *site_code* of all bookings:

```
SELECT Tour_id, Site_code
FROM BOOKING
```

On our database from Example 1, the standard answer to this query consists of the records that are written in normal font below:

Tour_id	Site_code
BRF	null
BRT	OR
null	BB
RF	**BB**

Tour_id	Site_code
null	BR
RF	null
RF	**BB**
RF	**OR**

Partial semantics suggests to add the records written in bold font, as these have corresponding parents in the TOUR table. As illustrated above, users benefit from highlighting non-standard answers and placing them directly below the records in the standard answer from which they originate.

Summary. Our intelligent query and update services exploit partial semantics to minimize information incompleteness. This results in higher quality data, better data-driven decision making, and more competitive organizations. Both services complement each other: Fewer choices for imputation mean fewer choices for intelligent updates, and more choices for imputation mean more non-standard query answers users can benefit from. So, whichever case we encounter at least one of the services is useful. It is beyond the scope of this paper to go deeper into the specific application of our proposed services. Instead, we see them as strong drivers to investigate the second open question, that is, whether partial semantics can be enforced efficiently.

6 Operational Requirements

This section examines the two main operational requirements for enforcing partial semantics. First, we propose implementation details in the form of triggers on child and parent schemas. Next we discuss five different index structures. Their impact on the performance of enforcing partial referential integrity will be analyzed in subsequent sections.

6.1 Triggers for Partial Referential Integrity

Foreign keys commonly enforce simple semantics in current database management system implementations. Our next goal is to define triggers that enforce partial referential integrity under updates. For that purpose we first propose a trigger on the referencing child schema C_S. This trigger will enforce partial referential integrity for $\langle$*Insert*$\rangle$ and $\langle$*Update*$\rangle$ modifications on any child table C. The referential action we uniformly consider in our experiments is $\langle$*SET NULL*$\rangle$.

We have designed a platform on `www.sqlkeys.info` which generates triggers for enforcing partial semantics on any arbitrary database with foreign keys up to size five. Below is the SQL code for a trigger that implements a referential integrity constraint on $n = 3$ columns:

```
**Trigger on $C_S$:**
BEFORE INSERT ON $C_S$ FOR EACH ROW
Declare $msg$ varchar(80);
If (new.$f_1$ is not null and new.$f_2$ is not null and
    new.$f_3$ is not null) then
  if not exists (select * from $P_S$ where ($k_1$=new.$f_1$
  and $k_2$=new.$f_2$ and $k_3$=new.$f_3$)) then
  set $msg$ =‘No reference is found, enter a valid value’;
  signal sqlstate ’02000’ set message_text = $msg$;
  end if;
Elseif (new.$f_1$ is not null and new.$f_2$ is not null and
  new.$f_3$ is null) then if not exists(select * from $P_S$
  where ($k_1$=new.$f_1$ and $k_2$=new.$f_2$) LIMIT 1) then
  set $msg$ =‘No reference is found, enter a valid value’;
  signal sqlstate ’02000’ set message_text = $msg$;
  end if;
Elseif (new.$f_1$ is not null and new.$f_3$ is not null and
  new.$f_2$ is null) then if not exists (select * from $P_S$
  where ($k_1$=new.$f_1$ and $k_3$=new.$f_3$) LIMIT 1) then
  set $msg$ =‘No reference is found, enter a valid value’;
  signal sqlstate ’02000’ set message_text = $msg$;
  end if;
Elseif ... /* similar for all $2^n - 1$ possible states */
End if;
End;
```

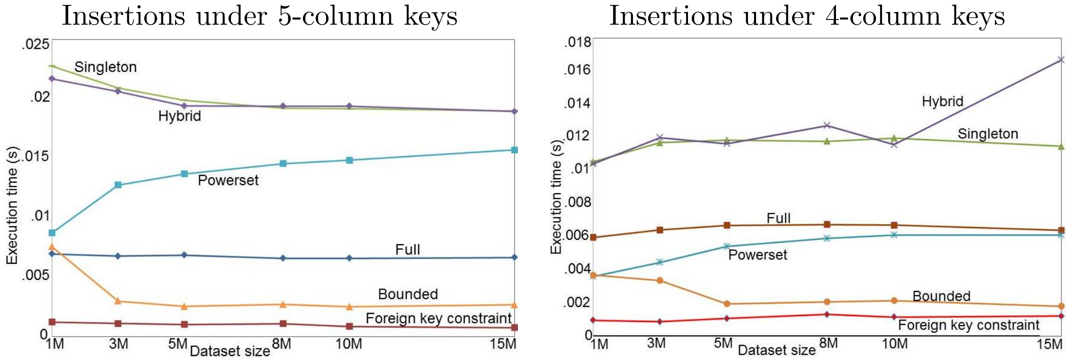
For ⟨*Delete*⟩ and ⟨*Update*⟩ modifications on the parent schema P_S another trigger is defined. The SQL code of the trigger in the case of $n = 3$ columns is as follows:

```
**Trigger on $P_S$:**
AFTER DELETE ON $P_S$ FOR EACH ROW
Update $C_S$ set $f_1$ =null, $f_2$ =null, $f_3$ =null where
     (old.$k_1$ = $f_1$ and old.$k_2$ = $f_2$ and old.$k_3$ = $f_3$);
If   exists (select * from $C_S$ where ($f_2$ is null and $f_3$
    is null and old.$k_1$ = $f_1$) limit 1) and not exists
   (select * from $P_S$ where old.$k_1$ = p.$k_1$ limit 1)
then   update $C_S$ set $f_1$ =null, $f_2$ =null, $f_3$ =null
   where (($f_2$ is null or $f_3$ is null) and old.$k_1$ = $f_1$);
end if;
If   exists(select * from $C_S$ where ($f_1$ is null and
   $f_3$ is null and old.$k_2$ = $f_2$) limit 1) and not exists
   (select * from $P_S$ where old.$k_2$ = p.$k_2$ limit 1)
then   update $C_S$ set $f_1$ =null, $f_2$ =null, $f_3$ =null
   where (($f_1$ is null or $f_3$ is null) and old.$k_2$ = $f_2$);
end if;
If   exists (select * from $C_S$ where ($f_1$ is null and $f_2$
   is null and old.$k_3$ = $f_3$) limit 1) and not exists
   (select * from $P_S$ where old.$k_3$ = p.$k_3$ limit 1)
then   update $C_S$ set $f_1$ =null, $f_2$ =null, $f_3$ =null
   where (($f_1$ is null or $f_2$ is null) and old.$k_3$ = $f_3$);
end if; /* similar for all $2^n - 1$ possible states */
End;
```

Table 1: Execution Time (s) for Insertion with a 5-Column Foreign Key

Data Set Size	Partial Semantics												Simple Semantics	
	No Index		Full		Singleton		Hybrid		Powerset		Bounded			
	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max
15M	1.98	9.11	0.0066	0.109	0.019	0.187	0.0189	0.156	0.0157	0.312	0.0026	0.078	0.00076	0.046
10M	0.69	6.63	0.0065	0.093	0.019	0.188	0.0194	0.187	0.0148	0.343	0.0025	0.047	0.00085	0.031
8M	0.57	5.21	0.0066	0.094	0.019	0.187	0.0194	0.218	0.0145	0.312	0.0027	0.063	0.00109	0.031
5M	0.33	2.55	0.0068	0.078	0.019	0.172	0.0194	0.187	0.0137	0.125	0.0025	0.063	0.00103	0.031
3M	0.24	1.51	0.0067	0.094	0.021	0.234	0.0206	0.203	0.0127	0.297	0.0029	0.078	0.00110	0.047
1M	0.15	0.57	0.0069	0.078	0.022	0.156	0.0217	0.156	0.0087	0.093	0.0075	0.125	0.00123	0.016

Figure 3: Performance Trends for Enforcing Partial Semantics under Insertions and Different Indices



6.2 Index Structures

One feature that significantly affects the system behavior is the index structure applied to the referenced and referencing tables. An appropriate index structure can optimize searches and improve the performance by several reads in one scan.

0) *No*: No index is defined. This option is a baseline for judging the performance of actual indices.

1) *Full*: One index is defined on $[k_1, \ldots, k_n]$ over P_S, and one index on $[f_1, \ldots, f_n]$ over C_S. Full enforces simple semantics [22]. Full might not improve partial referential integrity enforcement since a null marker in the foreign key may lead to a complete scan on all parent key values from the leftmost to the rightmost column [9].

2) *Singleton*: One index is defined for each k_i over P_S, and for each f_i over C_S, for $i = 1, \ldots, n$, resulting in $2n$ indices. With individual access to each column this approach is expected to boost the performance of enforcing partially-defined foreign keys [9].

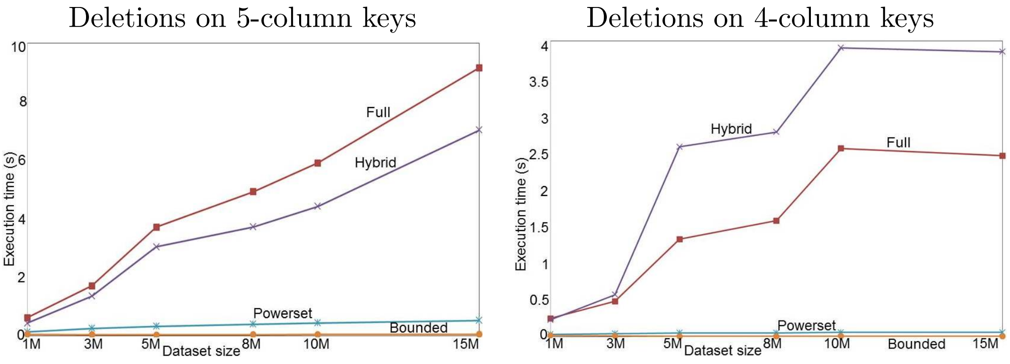
3) *Hybrid*: One index is defined for each k_i over P_S, and a single index on $[f_1, \ldots, f_n]$ over C_S, resulting in $n+1$ indices. According to [9], Hybrid takes advantage of both Full and Singleton options, and the authors conjectured that Hybrid best supports partial semantic.

4) *Powerset*: One index is defined on each non-empty subset of P_S, and on each non-empty subset of C_S, resulting in $2^{n+1} - 2$ indices. Powerset shows the impact of

Table 2: Execution Time (s) for Deletion with a 5-Column Foreign Key

Data	Partial Semantics												Simple	
Set	No Index		Full		Singleton		Hybrid		Powerset		Bounded		Semantics	
Size	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max
15M	286.04	824.7	9.16	203.1	110.45	813.9	7.03	142.20	0.531	0.967	0.057	0.296	0.0026	0.047
10M	201.38	480.3	5.91	151.3	109.00	651.7	4.42	100.34	0.442	0.904	0.047	0.234	0.0016	0.046
8M	128.04	393.7	4.93	119.8	93.32	572.8	3.72	82.41	0.401	0.92	0.042	0.218	0.0017	0.062
5M	85.79	255.48	3.7	86.15	39.2	337.1	3.04	67.76	0.330	0.79	0.037	0.188	0.0015	0.047
3M	52.65	138.76	1.7	63.43	15.4	143.3	1.36	41.52	0.255	0.655	0.041	0.156	0.0015	0.047
1M	19.61	41.77	0.62	13.58	3.5	25.4	0.44	8.70	0.140	0.577	0.057	0.266	0.0011	0.031

Figure 4: Performance Trends for Enforcing Partial Semantics under Deletions and Different Indices


Deletions on 5-column keys

Deletions on 4-column keys

having all possible indices available.

5) *Bounded*: One index is defined on $[k_1, \ldots, k_n]$ and one index for each k_i over P_S, and one index is defined on $[f_1, \ldots, f_n]$ and one index for each f_i over C_S, for $i = 1, \ldots, n$, resulting in $2n + 2$ indices. This structure combines Full, Singleton, and Hybrid, and reduces Powerset to just the singletons (lower bound) and the full subset (upper bound) instead of all subsets. Bounded outperforms all other structures in our experiments.

7 Experiments on Synthetic Data

We report on some of our experiments to evaluate the performance of enforcing partial semantics. Experiments were run on a Dell Latitude E5530, Intel core i7, CPU 2.9GHz with 8GB RAM. The operating system was Windows 7 Professional, Service pack 1 on a 64-bit operating system. The DBMS we used was MySQL version 5.6.

7.1 Description

All experiments involved two table schemata P_S and C_S and the foreign key $[f_1, \ldots, f_n] \subseteq P_S[k_1, \ldots, k_n]$ on C_S. Here, n varied between 2 to 5 to focus on the constraints that mostly occur in practice. For $n = 1$ there is no difference between simple and partial semantics. Parent table P and child table C were populated with synthetic data sets of various sizes between 1M and 15M tuples in P and 1.5 times as many tuples in C, respectively.

Columns of the candidate key $\{k_1, \dots, k_n\}$ on P_S did not feature `null`, while null markers did occur in the foreign key columns $\{f_1, \dots, f_n\}$ in C. This allows us to gain insights on the foreign keys that mostly occur in practice. Each non-empty subset S of $\{f_1, \dots, f_n\}$ had the same number of tuples in C which featured null markers in all columns in S and no null markers in any column outside of S. This also meant that the order of columns in a compound index does not matter in the experiments. We also run experiments where 50% and 80% of the tuples in C featured null markers in the foreign key columns, but the performances were very similar in each case. The performance of enforcing $[f_1, \dots, f_n] \subseteq P_S[k_1, \dots, k_n]$ was measured as the average time to insert tuples in C and to delete tuples from P, respectively, exploiting the triggers and different index structures from Section 6. For each data set and each index structure, the average was taken over 5,000 deletes and 5,000 inserts, respectively. We also compared the performance against that of simple semantics, enforced by built-in foreign keys. Note that execution times include the time for the trigger and the maintenance of the index structure. All reported experiments used BTrees. Applying Hash indices to our experiments resulted in similar outcomes, showing worse performance with minor exceptions. For these reasons we do not further comment on Hash indices here.

7.2 Impact of Indices

The impact of the index structures from Section 6 on the performance of enforcing partial semantics is the central contribution of our work. Tables 1 and 2 show the times to perform insertions into C and deletions from P, respectively, on the different data sets and where $n = 5$. These times are illustrated in Figures 3 and 4, respectively, along with the results for the same tests where $n = 4$. As expected, the use of indices leads to tremendous time savings for insertions and deletions. Our experiments confirm Härder and Reinhart's calculations that *Hybrid* achieves a performance similar to that of *Singleton* under insertions, and to that of *Full* under deletions. That is, it combines the performance gains of *Singleton* over *Full* under insertions, and the gains of *Full* over *Singleton* under deletions [9]. However, *Powerset* performs better than *Hybrid* under both insertions and deletions, and *Bounded* is the clear winner for both operations. On the largest data set with the largest foreign key size, for example, *Bounded* performs insertions/deletions on average about 7/123 times faster than *Hybrid*. The difference is considerable: the average time for deletions is 7.03s for *Hybrid* while it is 57ms for *Bounded*. *Bounded* is 6/9 times faster than *Powerset* on the largest data set. The performance gain of *Bounded* over *Powerset* shows that the additional time for maintaining further indices in *Powerset* and to choose the index from all the options in *Powerset* outweighs the time gains for the actual operations by *Powerset* in comparison to *Bounded*. Due to space restrictions we only present the results for 5-column foreign keys. For 3-column and 4-column foreign keys the index structures show similar behavior as the ones presented for 5-column foreign keys. Table 3 illustrates the results of applying *Bounded* and *Hybrid* on a synthetic data set of size 100M with a 5-column foreign key.

Exception. *Hybrid* shows the best performance when $n = 2$ and the data set size is large. For example, on the largest data set it takes 2.8/10.2ms for insertions/deletions, while these times increase to 4.3/11.5ms on *Powerset*, see Figure 5. Note that *Powerset*

Table 3: Execution Time (s) for 100M Data Set under some Index Structures and 5-Column Foreign Key

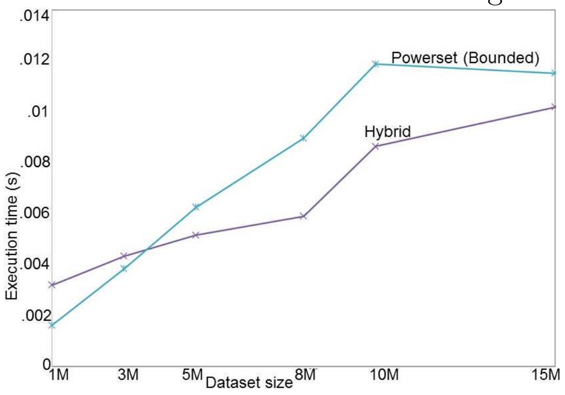
	Insertion		**Deletion**	
	Mean	**Max.**	**Mean**	**Max.**
Hybrid	0.013	0.156	39.74	976.23
Bounded	0.0027	0.063	0.085	0.281
Simple S.	0.001	0.047	0.0021	0.062

Table 4: Times (s) for Loading Data and Building Indices for 5-Column Key

Data Size	**No Index**		**Full**		**Singleton**		**Hybrid**		**Powerset**		**Bounded**		**Foreign key**
	C	***P***	***C***	***P***	***C***	***P***	***C***	***P***	***C***	***P***	***C***	***P***	
15M	107.2	69.1	403.7	96.3	297.2	198.3	405.8	200.9	5269.2	8716.8	622.6	234.5	587.9
10M	69.0	43.9	234.5	63.8	199.2	132.9	242.6	134.7	2875.4	5305.8	366.8	156.7	262.0
8M	56.4	36.0	172.3	51.8	159.3	107.6	171.7	106.0	2061.3	4073.4	275.5	120.5	180.3
5M	42.2	22.8	99.3	32.2	95.2	67.0	94.2	63.9	1016.3	2298.1	160.5	77.6	100.9
3M	20.5	13.3	53.5	19.5	63.9	41.4	54.0	40.7	522.6	1162.0	96.2	47.1	53.6
1M	7.9	4.9	16.7	7.4	20.1	13.6	15.3	13.3	142.6	151.1	28.2	14.72	16.9

and *Bounded* coincide when $n = 2$.

Figure 5: Deletions for 2-Column Foreign Keys



Simple semantics. Of course, it takes longer to enforce partial than simple semantics. However, the additional time becomes feasible under *Bounded*. For example, for insertions/deletions on the 15M data set with a 5-column foreign key, partial semantics is enforced by about 2.6/57ms, respectively. This takes 3.4/22 times longer than for simple semantics, while it takes 22/2703 times longer than for simple semantics using *Hybrid*. Even on a data set with 100M tuples, inserts and deletions can be processed within 2.7ms and 84.8ms, respectively, using *Bounded*. This confirms the feasibility of enforcing partial semantics on even large foreign key sizes.

7.3 Index Building

While the time to load data and build the indices is just a *one-time* cost, we still include it in our analysis. Table 4 shows the time taken to load data and build the indices on all data sets. Not surprisingly, the more indices are defined the longer it takes to build them. Building *Powerset* is thus time-consuming: more than 3hrs and 53mins on the largest data set, while *Hybrid* takes just over 10mins. *Bounded*, with twice as many indices as *Hybrid*, takes about 14mins and 30s. The foreign key index is built in 9mins and 47s.

7.4 Impact of Update Size

We performed some experiments with transactions, that is, atomic sets of update operations. The reported experiments were conducted for the 5-column foreign key on the data set with 15M tuples. The first experiment featured an insert of 5,000 tuples into C, and the second experiment a deletion of 2,000 tuples from P. The results are illustrated in Table 5. The transaction with 5,000 inserts takes just under 7s with *Bounded*, and nearly 90s with *Hybrid*. For the transaction with 2,000 deletions it takes over 148mins with *Hybrid*, and just under 111s with *Bounded*.

Table 5: Execution Time (s) of Transactions under Index Structures on Data Set with 15M Tuples

	5000 Insertions	**2000 Deletions**
Full	28.533	13413.16
Singleton	90.137	59191.81
Hybrid	89.65	8922.6
Powerset	102.81	605.71
Bounded	6.973	110.37
Simple S.	0.811	32.92

7.5 Extended Tests and Analysis

Deletions. Tables 2 and 5 show the poor performance of *Hybrid* on delete actions, and that it is overcome by *Bounded*. We will now analyze how *Bounded* achieves this. For that purpose, we exploited MySQL's explain statement which shows the optimizer's plan in executing the statements of each test. Recall that *Hybrid* has only one compound index over all foreign key columns in C_S. When we *Delete* from P, this means that one scan through all tuples is required to apply the referential action to children that feature `null` on the left most column. However, the referential action is only needed when there is no alternative parent for these children. Establishing that referential action must be applied to children whose only parent has been deleted is therefore poorly supported by *Hybrid*. We validated this observation by applying a deeper analysis to the experiment with our data set of 10M tuples. For this purpose, we call a parent *unique* when it has only children for which it is the only parent. Therefore, referential actions apply to all children of a unique parent. Otherwise, the parent is called *non-unique*. Table 6 shows the average execution time for deleting unique and non-unique parents when *Hybrid* is applied.

Table 6: *Hybrid* for Deletions, Average of Execution Times

Key size	**Non-unique Parent**	**Unique Parent**
5-Column keys	0.525s	40.47s
4-Column keys	2.37s	18.68s
3-Column keys	0.022s	17.59s

Table 7 shows the average execution time for deleting non-unique and unique parents when *Bounded* is applied.

Table 7: *Bounded* for Deletions, Average of Execution Times

Key size	**Non-Unique Parent**	**Unique Parent**
5-Column keys	0.051s	0.005522s
4-Column keys	0.00976s	0.00305s
3-Column keys	0.00348s	0.00167s

These results show that *Hybrid* performs particularly poor when deleting unique parents. The same analysis applies to transactions. In fact, only 3% of the deleted parents were unique. With *Hybrid* they occupied 145mins of the overall time of 148mins, but with *Bounded* they occupied only 0.5s of the overall time of 111s, refer to Table 13.

This poor performance of *Hybrid* can be avoided by adding to *Hybrid* one index on each foreign key attribute f_i on C_s. The resulting structure consists of $2n + 1$ indices in total, and we refer to it by *Hybrid+nSingle*. Table 8 shows the average execution time for deleting non-unique and unique parents when *Hybrid+nSingle* is applied.

Table 8: *Hybrid+nSingle* for Deletions, Average of Execution Time

Key size	**Non-Unique Parent**	**Unique Parent**
5-Column keys	0.413s	0.0061s
4-Column keys	2.29s	0.003s
3-Column keys	0.0237s	0.00233s

Another index structure that we have also tested is *Hybrid+Compound*, which consists of *Hybrid* plus one index over the key columns of P_S. *Hybrid+Compound* has therefore a total of $n + 2$ indices. For deletions, the additional index improves the search for children which are not null in the leftmost columns. This is demonstrated by comparing the results for *Non-Unique Parents* in Tables 7 and 8.

Figure 6 illustrates how well *Hybrid+nSingle* and *Hybrid+ Compound* perform deletions in comparison to the other indices. Clearly, the performance boost of *Bounded* over *Hybrid* for deletions is mainly due to adding *nSingle*.

Insertions. Figure 7 illustrates how well *Hybrid+nSingle* and *Hybrid+Compound* perform insertions in comparison to the other indices. Clearly, the performance boost of *Bounded* over *Hybrid* for insertions is mainly due to adding *Compound*.

Figure 6: Performances under Deletions with 5-Column Foreign Keys

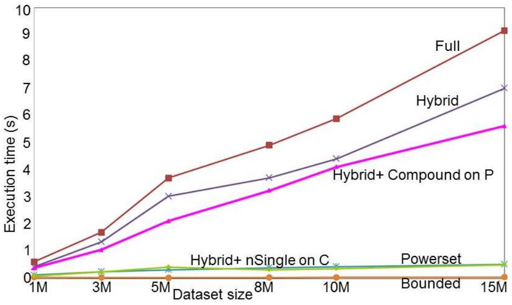
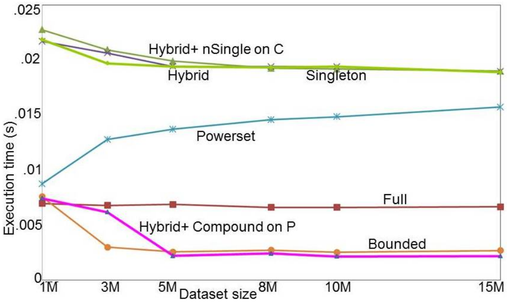
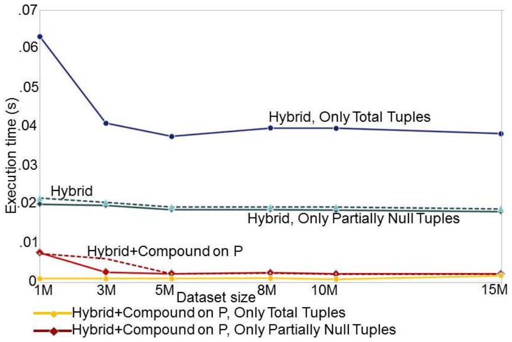


Figure 7: Performance under Insertions with 5-Column Foreign Keys



A deeper analysis confirms our intuition that *Hybrid* performs particularly poorly when inserting tuples that have only total foreign key values. Figure 8 breaks down the performance of *Hybrid* and *Hybrid+Compound* into insertions of tuples with only total foreign keys and those that are partially `null`.

Figure 8: Performance under Insertions with 5-Column Foreign Keys



Bounded is the only index structure that performs well under insertions and deletions, since it does not suffer from poor performance for insertions like *Hybrid+nSingle* and for deletions like *Hybrid+Compound*. In comparison to *Hybrid+nSingle* the better performances are achieved under negligible additional costs for building the indices, see Tables 11 and 12.

Transactions. Table 13 shows how the new index structures perform in transactions. *Hybrid+Compound* performs best for insertions but takes 149mins for deleting 2000 parents, while *Bounded* takes just 110s. *Hybrid+nSingle* is the runner-up to *Bounded* for

Table 9: Detail of the tested TPC databases

Database	**Parent table**	**Child table**	**Foreign key**
TPC-H	Partsupp Test 1: 0.8M records Test 2: 8M records	Lineitem 6M records 25% Null 60M records 60% Null	[*partkey*, *suppkey*] ⊆ Partsupp[*partkey*, *suppkey*]
TPC-C	Customer (90K records)	Orders (0.13M records) 55% Null	[*O-W-ID*, *O-D-ID*, *O-C-ID*] ⊆ Customer[*C-W-ID*, *C-D-ID*, *C-ID*]
TPC-C	Orders (0.13M records)	Orderline (1.3M records) 20% Null	[*OL-W-ID*, *OL-D-ID*, *OL-C-ID*] ⊆ Orders[*O-W-ID*, *O-D-ID*, *O-ID*]
Gene Ontology (GO) database	Term2term (T) (80k records)	Term2term-metadata (TT) (2200 records) 85% Null	[*relationship-type-id*, *term1-id* ,*term2-id*] ⊆ [*relationship-type-id* , *term1-id*, *term2-id*]

Table 10: Execution Time(ms) to Enforce Partial Referential Integrity on Benchmark Databases

	No Index	**Full**	**Singleton**	**Hybrid**	**Powerset**	**Bounded**	**Simple S.**
TPC-H Test 1: Insert into Lineitem	161	1.8	1.6	1.3	1.1	-	1.06
Delete from Partsupp	10.92 (s)	6.1	148	5.6	4.3	-	2
TPC-H Test 2: Insert into Lineitem	1.97s	3.1	1.1	0.76	1.09	-	0.88
Delete from Partsupp		19.7	151.72	5.94	14.79	-	3.10
TPC-C: Insert into Orders	14.5	2.73	2.95	2.84	1.28	1.02	0.81
Delete from Customer	498	11.8	120.3	10.15	2.44	2.47	0.6
TPC-C: Insert into Orderline	9.23	2.44	1.9	2.25	1.62	1.31	1.21
Delete from Orders	2.92(s)	18.6	1.05(s)	15.58	3.78	3.52	1.285
GO Database: Insert into TT	15.6	6.16	1.31	1.16	2.37	1.3	1.12
Delete from T	54.2	2.11	17.35	12.6	2.4	1.87	1.06

Table 11: Index Building (IB) and Execution Time of *Bounded*

Dataset Size	**IB for *C* (s)**	**IB for *P* (s)**	**Insert Ave. (s)**	**Delete Ave. (s)**
15M	622.569	234.532	0.002678	0.0578
10M	366.821	156.73	0.00251	0.047189
8M	275.513	120.542	0.00271	0.0425
5M	160.463	77.563	0.00254	0.03723
3M	96.237	47.019	0.002988	0.04189
1M	28.173	14.742	0.00757	0.05729

Table 12: Index Building (IB) and Execution Time of *Hybrid+nSingle*

Dataset Size	**IB for *C* (s)**	**IB for *P* (s)**	**Insert Ave. (s)**	**Delete Ave. (s)**
15M	636.702	207.029	0.0189	0.5135
10M	367	136.204	0.0194	0.37
8M	282.409	109.045	0.01934	0.3316
5M	162.6	66.81	0.0194	0.43
3M	92.212	40.68	0.0197	0.2481
1M	28.25	13.26	0.02186	0.10149

deletions, but performs poorly for insertions.

Table 13: Execution Time (s) of Transactions under Index Structures

	5000 Insertions	**2000 Deletions**	
		Unique *p*	**Others**
Hybrid	89.65	8753.2	169.4
Hybrid+Compound	6.26	8830.8	104.05
Hybrid+nSingle	90.78	0.515	167.43
Bounded	6.973	0.499	109.9

8 Benchmark Data

We extend our analysis of enforcing partial semantics to some benchmark and real-world data. For that purpose, we have run experiments on one two-column foreign key from the TPC-H database and two three-column foreign keys from the TPC-C database (`www.tpc.org`). In addition, we have tested one three-column foreign key from the Gene Ontology (GO) database (`www.geneontology.org/GO.database.shtml`). Table 9 shows the details of the foreign keys. Here, the data set size for test 1 on TPC-H was 1.43GB, and for test 2 it was 10GB; for TPC-C it was 0.39GB, and for the GO database it was 100MB. Applying the "Missing at Random" mechanism from [23], null markers were introduced randomly and spread evenly between the foreign key columns.

We have tested the TPC-H benchmark with two different data set sizes (0.8M and 8M tuples). Note that *Powerset* and *Bounded* coincide on 2-column foreign keys and thus on the experiments with TPC-H. The results of enforcing partial and simple semantics on these databases are shown in Table 10. The performances rank very similar to those on the synthetic data sets. Our results on TPC-H confirm the observed changes on the performance of *Hybrid* and *Powerset* on larger data sets with 2-column foreign keys, see Figure 5. The TPC-C data set with the 3-column foreign keys confirms our result with the synthetic data sets that *Bounded* outperforms *Hybrid* by a factor of 2 for insertions and by 5 for deletions. Similar results have been observed on the Gene Ontology database. The best times for enforcing partial semantics are never over 6ms while the times for enforcing simple semantics are never over 4ms.

9 Conclusion and Future Work

Even two decades after its introduction to the SQL standard there are no database management systems with built-in support for partial referential integrity. Härder and Reinhart had argued on the operational level that partial semantics is too costly to implement [9]. They invited more research on its motivation and its costs at the systems level [9]. In this paper we addressed both questions. Firstly, we proposed intelligent update and query services that impute missing data values by exploiting partial semantics. In particular, we discussed how these services reduce information incompleteness in the database and suggest possible values by which null markers can be replaced in query answers. Secondly, we proposed two main operational requirements to enforce and speed up the enforcement of partial semantics, and conducted several performance tests in MySQL. Our tests were targeted at foreign keys that occur commonly in practice. These have very rarely more than five columns and reference candidate keys without null marker occurrences. Permitting occurrences of `null` in referenced candidate keys only affects our results marginally. Our first major finding confirms on the system level what Härder and Reinhart had calculated on the operational level: (1) The performance of the *Hybrid* index structure matches that of the *Singleton* index structure for insertions, and that of the *Full* index structure for deletions. (2) *Hybrid* is the best candidate to support `MATCH PARTIAL`, but only for 2-column foreign keys. We proposed here the new index structure *Bounded* for foreign keys with more than two columns. Our second major finding is that *Bounded* outperforms *Hybrid* for foreign keys with more than two columns. For example, for a 5-column foreign key on a data set with 15M tuples and a fair distribution of null markers, *Bounded* performed 7 times better for insertions and 123 times better for deletions. The better performance was confirmed on other synthetic data sets, for transactions, for two TPC-C data sets and a three-column foreign key on the Gene Ontology database. The only trade-off we found concerns the loading time of the data set and building of the indices, which essentially were 1.5 times more in comparison to *Hybrid*. Our third major finding is that the performance boost of *Bounded* for deletions results from adding one index to *Hybrid* on each foreign key column, and the performance boost of *Bounded* for insertions results from adding one index on the compound key to *Hybrid*. Overall, the enforcement of partial semantics with *Bounded* was never slower than 300ms for any atomic operation with a 5-column foreign key, while the enforcement of simple semantics was never slower then 62ms. Based on our results, we conclude that partial referential integrity can be enforced efficiently for real-life foreign keys, which opens up new applications such as intelligent queries and intelligent updates that lead to better quality data and better data-driven decision making. These applications do not apply to simple referential integrity. Our results demonstrate the benefits of partial semantics, and *Bounded* offers a principled approach to indexing foreign keys with partial semantics that can be added on top of existing databases in a non-intrusive fashion.

We hope our research will ignite future work on this topic. Certainly there are many interesting questions that a single paper cannot address, but which should be pursued in future work to unlock many potential benefits. Other indexing options can be studied, for example the combination of compound indices over key attributes and multidimensional access paths at the system level. An index option including $2n$ compound n-ary indices over the referenced key attributes $[k_i, \ldots, k_n, k_1, \ldots, k_{i-1}]$ and referencing key attributes $[f_i, \ldots, f_n, f_1, \ldots, f_{i-1}]$, for $i = 1, \ldots, n$, supports partial match index look-ups by the different prefixes of the compound indices. However, our initial analysis shows that *Bounded* outperforms this index option in deletions of 3, 4 and 5-column foreign keys by more than 3 times on data sets with 15M tuples. The loading times of the data sets and building of the indices in *Bounded* are always between 1.5 to 4 times cheaper. Exploiting $2n$ compound indices over key attributes is not enough to support all the possible partial match queries. For instance when $n = 5$, defining 2×5 compound indices in different orders only supports 21 of 31 match queries. Another interesting avenue deals with the trade-off between query and enforcement issues. While our results show that enforcement and maintenance of partial semantics are unlikely bottlenecks on databases of enterpriselevel size (10GB in test 2 for TPC-H), future research should be aimed at guidelines for resolving conflicts between resources for query and update acceleration in schemata for big data. While our solution of implementing enforcement by database triggers and index primitives is appealing in several ways, future work may reveal potential performance gains that could be realized with an engine-level implementation. For instance, there may be custom index data structures that leverage partial and adaptive indexing methods, as well as a more streamlined trigger execution in order to improve enforcement costs. Furthermore, there are several techniques such as batching and shared execution across updates that apply within transactions, and could therefore optimize the enforcement of partial referential integrity in this context. Our proposed services of intelligent queries and updates each open up their own areas of future investigation. For updates it would be interesting to investigate how large numbers of choices for imputations can be represented or ranked, how logs of potential imputations can best be processed by data analysts, or how unsuccessful imputations can be reversed. For queries, it would be interesting to find re-writings of SQL queries that return not only standard but also non-standard answers that result from the application of partial semantics, and to investigate the overhead of such techniques. Information incompleteness is also inherent in other data models such as graphs, RDF, or XML. Finally, implication problems of inclusion dependencies under SQL semantics should be studied [10, 11].

References

[1] J. Bauckmann, Z. Abedjan, U. Leser, H. Müller, and F. Naumann. Discovering conditional inclusion dependencies. In X. Chen, G. Lebanon, H. Wang, and M. J. Zaki, editors, *CIKM*, pages 2094–2098. ACM, 2012.

[2] M. A. Casanova, R. Fagin, and C. H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *J. Comput. Syst. Sci.*, 28(1):29–59, 1984.

[3] M. A. Casanova, L. Tucherman, and A. L. Furtado. Enforcing inclusion dependencies and referential integrity. In F. Bancilhon and D. J. DeWitt, editors, *VLDB*, pages 38–49. Morgan Kaufmann, 1988.

[4] A. K. Chandra and M. Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM J. Comput.*, 14(3):671–677, 1985.

[5] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[6] C. J. Date. *Relational database writings: 1985-1989.* Addison-Wesley, 1990.

[7] V. Ganti and A. D. Sarma. *Data Cleaning: A Practical Perspective.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[8] T. Halpin and T. Morgan. *Information modeling and relational databases.* Morgan Kaufmann, 2010.

[9] T. Härder and J. Reinert. Access path support for referential integrity in SQL2. *The VLDB Journal*, 5(3):196–214, 1996.

[10] S. Hartmann, M. Kirchberg, and S. Link. Design by example for SQL table definitions with functional dependencies. *The VLDB Journal*, 21(1):121–144, 2012.

[11] S. Hartmann and S. Link. The implication problem of data dependencies over SQL table definitions. *ACM Trans. Database Syst.*, 37(2):13, 2012.

[12] D. S. Johnson and A. C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci.*, 28(1):167–189, 1984.

[13] M. Karlinger, M. W. Vincent, and M. Schrefl. Inclusion dependencies in XML: Extending relational semantics. In S. S. Bhowmick, J. Küng, and R. Wagner, editors, *DEXA*, volume 5690 of *LNCS*, pages 23–37. Springer, 2009.

[14] G. Lausen. Relational databases in RDF: Keys and foreign keys. In V. Christophides, M. Collard, and C. Gutierrez, editors, *SWDB-ODBIS*, volume 5005 of *LNCS*, pages 43–56. Springer, 2008.

[15] M. Levene and M. W. Vincent. Justification for inclusion dependency normal form. *IEEE Trans. Knowl. Data Eng.*, 12(2):281–291, 2000.

[16] S. Link and M. Memari. Static analysis of partial referential integrity for better quality SQL data. In J. Shim, Y. Hwang, and S. Petter, editors, *AMCIS*, pages 38–49. Association for Information Systems, 2013.

[17] J. Melton. ISO/IEC 9075-2: 2003 (SQL/foundation). ISO standard, 2003.

[18] J. C. Mitchell. The implication problem for functional and inclusion dependencies. *Information and Control*, 56(3):154–173, 1983.

[19] J. C. Mitchell. Inference rules for functional and inclusion dependencies. In R. Fagin and P. A. Bernstein, editors, *PODS*, pages 58–69. ACM, 1983.

[20] C. Ordonez and J. García-García. Referential integrity quality metrics. *Decision Support Systems*, 44(2):495–508, 2008.

[21] B. Thalheim. *Dependencies in relational databases.* Teubner, 1991.

[22] C. Türker and M. Gertz. Semantic integrity support in SQL: 1999 and commercial (object-) relational DBMSs. *The VLDB Journal*, 10(4):241–269, 2001.

[23] Y. Wen, K. B. Korb, and A. E. Nicholson. DataZapper: Generating incomplete datasets. In J. Filipe, A. L. N. Fred, and B. Sharp, editors, *ICAART*, pages 69–76. INSTICC Press, 2009.

[24] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. On multi-column foreign key discovery. *PVLDB*, 3(1):805–814, 2010.