In: The VLDB Journal, Vol. 5, No. 3, 1996, pp. 196-214.

Access Path Support for Referential Integrity in SQL2

Theo Härder, Joachim Reinert
University of Kaiserslautern
Dept. of Computer Science, P.O.Box 3049
67653 Kaiserslautern, Germany
Phone: +49 631 205 4030
Fax: +49 631 205 3558
e-mail: {haerder | jreinert} @informatik.uni-kl.de

Abstract

The relational model of data incorporates fundamental assertions for entity integrity and referential integrity. Recently, these so-called relational invariants were more precisely specified by the new SQL2 standard. Accordingly, they have to be guaranteed by a relational DBMS to its users and, therefore, all issues of semantics and implementation became very important. The specification of referential integrity embodies quite a number of complications including the MATCH clause and a collection of referential actions. In particular, `MATCH PARTIAL` turns out to be hard to understand and, if applied, difficult and expensive to maintain.

In this paper, we identify the functional requirements for preserving referential integrity. At a level free of implementational considerations, the number and kinds of searches necessary for referential integrity maintenance are derived. Based on these findings, our investigation is focussed on the question of how the functional requirements can be supported by implementation concepts in an efficient way. We determine the search cost for referential integrity maintenance (in terms of page references) for various possible access path structures. Our main result is that a combined access path structure is the most appropriate for checking the regular MATCH option whereas `MATCH PARTIAL` requires very expensive and complicated check procedures. If it cannot be avoided at all, the best support is achieved by a combination of multiple B*-trees.

1 Introduction

In his “historical” paper about the relational model of data [Co70], E. F. Codd informally introduced entity integrity and referential integrity as the “relational invariants” to be automatically guaranteed for each relation by a relational DBMS (system-enforced integrity). In the meantime, several attempts have been made to formalize these important data model properties thereby setting the semantics of these integrity assertions more precisely [Sh90]. Now, more than 20 years later, the new SQL2 standard [Sh90, SQL92] defines these relational invariants for the relational model in a uniform way, with the goal of making them mandatory for all relational DBMS.

At the level of DB schema design, the new standard provides powerful concepts for specifying key conditions as well as referential integrity conditions. Besides the primary key condition (PRIMARY KEY), key uniqueness can be maintained for multiple candidate keys using the UNIQUE option. Matching values of primary key and foreign keys are guaranteed by the FOREIGN KEY concept which may be endowed with different matching semantics by the MATCH clause. However, this clause is responsible for quite a number of complications which may burden the design. Furthermore, the specification of different referential actions provides some automatic means to maintain the referential integrity for the case that some update operation violates the matching conditions of keys related via referential integrity.

The implementation of these rich and powerful concepts may drastically influence the DBMS performance. Therefore, it seems to be an urgent task to investigate the system behavior at the operational level. For this purpose, the various aspects of system overhead caused by the services for maintaining the relational invariants have to be studied in detail. A prime contributor is the run-time overhead resulting from the various searches to locate tuples or keys to be checked or compared. Moreover, update costs involving tuples and access path data have to be considered, e.g., for referential actions. Furthermore, additional costs may result from locking, logging, and related services.

Our goal is to study the usefulness of various access path types for referential integrity support. We believe that this question should be investigated at a suitable level of abstraction in order to achieve a sufficient selectivity between different possible access path types and, at the same time, to avoid low-level modeling that may only provide a kind of artificial accuracy. Thus, we don’t want to step into the intricacies of multi-user operation and of detailed access path and operation modeling. We focus our investigation on the estimation of search overhead and the use of different kinds of access paths (in terms of page references). Obviously, searching embodies the lion’s share of the operational costs. For this reason, these costs may be considered as indicative for the entire checking costs.

For the performance analysis of the relational invariants, checking the existence of a key is a very important and frequent operation. Locating the key or the tuple often implies a search in a large data set. Since sequential scans cannot be tolerated for apparent reasons, we assume that a suitable index exists for every key to be specified by the options UNIQUE, PRIMARY KEY, and FOREIGN KEY. Checking the entity integrity and the UNIQUE option is conceptually very simple; for each of the specified keys, an index access has to be performed for insert and update operations to check the uniqueness of the related key values. Therefore, we only focus on the performance aspects of referential integrity, especially the influence of the MATCH clause.

The remainder of the paper is organized as follows. Sect. 2 discusses the concepts of referential integrity as specified in the SQL2 standard. In this framework, in Sect. 3 we outline the functions to be performed whenever a relation is modified and we identify the number and kind of searches required to accomplish this task. Sect. 4 investigates the suitability of various access path structures when used for these searches. Furthermore, we derive the search costs in terms of logical page references and compare the performance behavior of the chosen access path candidates. Finally, Sect. 5 contains our conclusions and proposals for future work.

2 The Concepts of Referential Integrity

In this section, we analyze the possibilities of the new SQL2 standard [SQL92] in the area of referential integrity. For this reason, we present a short outline of referential integrity and then introduce the syntactical clauses for referential integrity in SQL2.

2.1 Referential Integrity

The concepts of referential integrity, originally defined by Codd [Co70] and influenced by Date [Da81, Da90], are included in the new SQL2 standard which was accepted by ANSI and ISO in 1992. To recall: Referential integrity is an integrity constraint between a set $F = \{f_1,..., f_n\}$ of attributes (called foreign key) of a relation C (called child) and a set $K = \{k_1,..., k_m\}$ of attributes of a relation P (called parent). The structural constraints on a database schema S implied by a referential integrity constraint are:

- $n = m$
- For each i, the domain of f_i is the same as the domain of k_i
- K is the primary key of P.

A referential integrity constraint implies the following constraint on the instances of S:
For every tuple t_C of C, there exists a tuple t_P in P with $t_C|_F = t_P|_K$[1].
An exception to this general rule are null values to express unknown or inapplicable values. If some attribute of F in t_C has such a null value as its value, no counterpart for $t_C|_F$ is needed in P (other constraints than referential integrity may be violated, e.g. constraints regulating the applicability of null values). Some modifications of these semantics are discussed later in this section.

1. $t|_X$ denotes the projection of t onto the attributes in X, if t is a tuple of relation R and X a set of attributes of R. This is extended in a canonical way to $R|_X$.

Since a referential integrity constraint is a static constraint, it may be violated by user operations. There are six basic update operations involving one of the relations P or C. “Insert into P” and “Delete from C” do not lead to a violation of the referential integrity whereas the remaining four operations potentially do. We briefly review each of these operations.

Delete from P

A tuple deleted from P may have some children referencing this tuple. After the deletion, these children no longer have a parent anymore; hence, this operation may violate the referential integrity.

Update attribute k_i of P

At this stage of discussion, an update can be viewed as a deletion of the tuple with the old value and an insertion of the tuple with the new value.

Insert into C

If all attribute values of the attributes f_i of the inserted tuple are different from the null value, referential integrity requires the existence of a matching tuple in P. If this tuple does not exist, referential integrity is violated.

Update attribute f_j of C

Similar to the update of P mentioned above, at this level of discussion we can view an update by a deletion and an insertion.

2.2 Referential Integrity in SQL2

In this section, we introduce the syntax of SQL2 for referential integrity and include a short discussion of the various options for referential actions.

In the SQL2 standard, referential integrity constraints are defined when tables are created or altered. For this purpose, a subclause of the `create table` or the `alter table` statement referring to the child table C is used. The complete subclause is as follows:

```
FOREIGN KEY (<referencing columns>)
  REFERENCES <table name> (<referenced columns>)
  [MATCH {FULL | PARTIAL}]
  [ON UPDATE {CASCADE | SET NULL | SET DEFAULT | NO ACTION}]
  [ON DELETE {CASCADE | SET NULL | SET DEFAULT | NO ACTION}]
```

The `<referencing columns>` are the attribute names of the foreign key F in C. The `<referenced columns>` denote the attributes of the primary key K of the parent table `<table name>`. Instead of the primary key of a table P as the referenced group of attributes, SQL2 allows so-called candidate keys to be referenced. Codd has introduced the term “candidate key” as a group of attributes of a relation that allows each tuple of the relation to be uniquely identified by these attributes, i.e., the primary key is one of the candidate keys. But opposed to the primary key, the value of a candidate key may be partly undefined (null values). The implications of this extension will be discussed in Sect. 3.3.

The semantics expressible through the subclause `MATCH {FULL | PARTIAL}` specifies the interpretation of null values in the foreign key of a tuple t_C. In Sect. 3.2, we explain the special semantics achievable with this subclause.

The subclauses `ON UPDATE ...` and `ON DELETE ...` allow special treatments when referential integrity is violated by a user operation as discussed in the previous section. Given a DB state which fulfills referential integrity, only four out of the six operations may violate it. According to the SQL2 standard, the two operations "Insert into C" and "Update f_i of C" on a child are forbidden (backed out) if these would result in DB states where referential integrity is not fulfilled. Therefore, only the two operations ("Delete from P" and "Update k_i of P") on a parent relation are handled in a special way:

- `ON UPDATE`. If a key attribute referenced in a referential integrity constraint is updated in a tuple t_P, then the following actions are carried out depending on the specification in the schema:
  - `CASCADE`. The new values in the key are propagated to the referencing children.
  - `SET NULL`. The resp. foreign key attributes in referencing tuples t_C are set to the null value.
  - `SET DEFAULT`. The corresponding foreign key attributes in referencing tuples t_C are set to a default value (definable for each attribute in the schema).
  - `NO ACTION`. The referential action is delayed on relation C. Referential integrity remains violated and if no other operation takes place to correct the mismatch of the corresponding tuples t_C, the complete work of the transaction will finally be backed out. This happens either at the end of the statement (if the integrity checking is not deferred) or at transaction commit (if the integrity checking is deferred). A discussion of deferred integrity checking is beyond the scope of this paper.
- `ON DELETE`. If a tuple t_P is deleted, then the following actions are carried out depending on the specification in the schema:
  - `CASCADE`. The referencing children are also deleted.
  - `SET NULL`. The foreign key attributes of the children are set to the null value.
  - `SET DEFAULT`. The foreign key attributes of the children are set to the given default value.
  - `NO ACTION`. Nothing is done. Referential integrity remains violated and if no other operation takes place to correct this, the complete work of the transaction will be backed out.

There is another important referential action not introduced in the SQL2 standard, but in nearly all papers dealing with referential integrity: `RESTRICT` (or `RESTRICTED` depending on the author). The semantics of this referential action is to forbid any change (update or delete) of a parent tuple t_P as long as there are referencing child tuples t_C. Although this action is not in the SQL2 standard (but scheduled for SQL3 [SQL3]) we will include it in our discussion.

A problem of the referential integrity constraints as specified in SQL2 results from the possibility of interference when performing multiple referential actions on one tuple. That is, a straightforward implementation may lead to an indeterminism in the result of a user operation, i.e., an operation in single-user mode may cause different database states if the triggered referential actions are executed in different sequences on the same database state. The SQL2 standard prevents such indeterminism through the specification of a complex test carried out during the execution of the referential actions. A detailed discussion of this approach may be found in [Ma90, Re93]; it is beyond the scope of this presentation.

3 Functional Requirements

Since we want to support efficient integrity checking, a critical question is: "Which access patterns have to be supported?". In order to answer this question, we concentrate on the searches required for observing a referential integrity constraint. In principle, there will be no new operations; however, in contrast to traditional relational query processing where only *complex* queries result in *complex* evaluations, now such complex evaluations may be forced by *simple* queries.

3.1 Overhead of regular Referential Integrity

First we analyze the referential integrity in the simplest setting. Therefore, we introduce the following restrictions on the definition of referential integrity constraints in SQL2:

- The attributes of a foreign key are either not allowed to be null or foreign keys having null values for some attributes are not taken into account when checking referential integrity, i.e., special treatment of null values is not considered. This is expressible by the MATCH clause (see Sect. 3.2).
- The referenced group of attributes is the primary key of the referenced relation. Therefore, null values are excluded.

We now discuss the operations that may raise problems with referential integrity. As already said, this discussion is to determine the functional requirements to be met by a system for supporting referential integrity. Thus, we focus on the different query types which should be supported to achieve efficient constraint checking. As mentioned earlier, efficient checking of referential integrity requires some access paths to avoid (multiple) sequential scans on the relations. On the other hand, these access paths have to be maintained whenever the underlying relations are modified. Therefore, the search requirements for integrity checking are made up of two parts:

- The costs of locating the tuples or keys which allow the required check.
- The old and the new locations of the tuples or keys in the access path have to be selected, if update operations are necessary. In our scenario, maintenance always follows the constraint checking directly. Therefore, we assume that the old location of the keys is already known. Thus, the search costs for maintenance consist of the overhead to determine the new location of the keys in the access path if required[1] (e.g., if `ON DELETE SET DEFAULT` is specified).

1. Deferred checking or unusual cases of `NO ACTION` do not allow such an approximation.

In the following, we consider the test whether or not the parent key is unique as an integral part of referential integrity checking. Therefore, we include the necessary checks and maintenance actions into our requirements.

Insert into P

During the insertion of a tuple t into P, it must be checked whether or not the primary key of t is unique within P. To decide this question, a query may be issued to select all tuples with the same primary key as t. If the result of this query is not empty, then another tuple with the same primary key exists and hence t must not be inserted. We denote the type of the mentioned query as $(\mathbf{P}, \mathbf{E})_P$ to express that it is a **P**oint query (in the space of the keys) which tests the **E**xistence of one key. The P in the subscript denotes that the query is evaluated on the parent relation.

Delete from P

To delete a tuple t_P from the relation P, it is located and checked whether or not there are any related child tuples t_C. To locate t_P via an access path we need a query of type $(\mathbf{P}, \mathbf{T})_P$ (**P**oint query with one resulting **T**uple). The query types needed to test and maintain the children depend on the option specified in the schema:

- `CASCADE`. To access all children of t_P, a **P**oint query is issued in the foreign key space which results in a **S**et (there may be more than one child) of tuples t_C. We will denote this query type by $(\mathbf{P}, \mathbf{S})_C$. Together we obtain the abstract costs of $(\mathbf{P}, \mathbf{T})_P + (\mathbf{P}, \mathbf{S})_C$.
- `RESTRICT`. To test whether or not to perform the operation (Delete from P), the evidence of at least one child is sufficient. Therefore, the query type is $(\mathbf{P}, \mathbf{E})_C$. Hence, the complete operation results in cost $(\mathbf{P}, \mathbf{T})_P + (\mathbf{P}, \mathbf{E})_C$.
- `SET NULL`. As for `CASCADE` the children are selected through a query of type $(\mathbf{P}, \mathbf{S})_C$. In contrast to that case we have to update the children location in the access path because their foreign key is changed ($(\mathbf{P}, \mathbf{S})_C$). Therefore the costs are $(\mathbf{P}, \mathbf{T})_P + 2 \cdot (\mathbf{P}, \mathbf{S})_C$.
- `SET DEFAULT`. Compared to the `SET NULL` option, an additional query is necessary if the default values differ from the null value. In this case, the related children get a new fully defined foreign key; thus, it has to be tested whether or not the new (default) parent exists. This leads to another $(\mathbf{P}, \mathbf{E})_P$ query and to the entire overhead of: $(\mathbf{P}, \mathbf{T})_P + (\mathbf{P}, \mathbf{E})_P + 2 \cdot (\mathbf{P}, \mathbf{S})_C$.
- `NO ACTION`. This option is difficult to evaluate in general because of the following:
  - If some attributes of F serve as foreign key attributes in more than one referential integrity constraint simultaneously or if the integrity checking is deferred, other operations (initiated by the user or through other referential integrity constraints) may resolve the conflict introduced by the deletion of t_P. Hence, opposed to the other options (`CASCADE`, `SET NULL`, `SET DEFAULT` and `RESTRICT`) which guarantee referential integrity after the appropriate action is carried out, this

option will require an *explicit* integrity checking at some time in the future. The type of these queries is (**P**, **E**)$_P$[1].

- In all other cases, this option is identical to `RESTRICT` and therefore the query type is (**P**, **E**)$_C$.

The above discussion shows that an efficient evaluation of query type (**P**, **S**) is necessary for integrity checking. Sometimes an optimization by a (**P**, **E**) query is possible, but for simplicity reasons we do not elaborate on this in the subsequent sections and use the worst case (`SET DEFAULT`) which is made up of one (**P**, **T**)$_P$, one (**P**, **E**)$_P$ and two (**P**, **S**)$_C$ queries.

Update attribute k_i of P

The overhead of checking the referential integrity in this case consists of four parts (in the worst case): Firstly, the parent and all related children have to be located which requires (**P**, **T**)$_P$ + (**P**, **S**)$_C$. Secondly, the new value of K has to be checked for uniqueness. This is achieved through a query of type (**P**, **E**)$_P$ (as in the insert case). The third part may be needed if `SET DEFAULT` is specified, because the children change their parent (now it is the "default" parent) and the existence of this parent has to be checked with another (**P**, **E**)$_P$ query. Last not least, the foreign key of the children is changed and therefore the underlying access path is updated leading to a (**P**, **S**)$_C$ query. Thus, the worst case consists of (**P**, **T**)$_P$, 2·(**P**, **E**)$_P$ and 2·(**P**, **S**)$_C$, which is also the sum of a delete and an insert.

Insert into C

If a tuple t_C is inserted into the relation C, a check is required whether there is a matching parent tuple t_P or not. The insertion fails if no parent exists. The checking overhead consists of a query of the type (**P**, **E**)$_P$. Furthermore, the access path of the foreign key has to be maintained. Hence, we have to locate the insertion point of the new child requiring a query of type (**P**, **T**)$_C$.

Delete from C

The tuple to be deleted from the set of children has to be located in the access path. This requires a query of the type (**P**, **T**)$_C$.

Update attribute f_j of C

If an attribute of the foreign key is updated, the existence of a parent tuple for the new foreign key value must be checked. This is a query of type (**P**, **E**)$_P$. The maintenance of the access path requires two queries of the type (**P**, **T**)$_C$ to "move" the child from the old to the new location.

In this section, we have analyzed the query types of regular referential integrity constraints. Up to now, we did not mention multi-attribute foreign keys explicitly because (with the preconditions about null val-

---

1. At this point in time, the database may have gone through multiple changes and without any internal bookkeeping about referential integrity it may be better to check it on a relation basis, i.e., check for all tuples in C whether there is a tuple in P with matching primary key.

ues stated at the beginning of this section) such foreign keys can be simulated by one super-attribute composed of the single attributes (subsequently called compound attribute). In the following, we will analyze the semantics of null values in connection with `MATCH PARTIAL`.

3.2 Overhead of the **`MATCH PARTIAL`** clause

The definition of the `MATCH` predicate serves as the basis of the semantics of the `MATCH PARTIAL` subclause for referential integrity.

*The **`MATCH`** Predicate*

The `MATCH` predicate tests a tuple t against a set of tuples M: a group of attributes of t is compared tuple-by-tuple with a related group of attributes in set M (the attribute domains have to be pairwise comparable). The definition of this predicate allows the optional specification of `PARTIAL` or `FULL`. Given $t|_{<f1,...,fn>}$ `MATCH[{FULL | PARTIAL}]` $M|_{<k1,..., kn>}$ and a tuple m of M, the result of this comparison is as follows:

- No option specified
  - If some attribute f_i of t has the value null, then `MATCH` results in `TRUE`.
  - If no attribute f_i of t has the value null and $t.f_i = m.k_i$ $(1 \le i \le n)$, then `MATCH` results in `TRUE`.
  - Otherwise `MATCH` results in `FALSE` (for this tuple m).
- `FULL` is specified
  - If all values f_i are null, then `MATCH` results in `TRUE`.
  - If no value f_i is null and $f_i = k_i$ $(1 \le i \le n)$, then `MATCH` results in `TRUE`.
  - Otherwise `MATCH` results in `FALSE` (for this tuple $<k_1,..., k_n>$).
- `PARTIAL` is specified
  - If all values f_i are null, then `MATCH` results in `TRUE`.
  - If $f_i = k_i$ holds for all defined values f_i, then `MATCH` results in `TRUE`.
  - Otherwise `MATCH` results in `FALSE` (for this tuple $<k_1,..., k_n>$).

Without any option or the option `FULL` specified, a null value in one attribute f_i determines the result of the whole evaluation of the `MATCH` predicate. If `PARTIAL` is specified, null values in f_i are treated as don't-care values. There is no symmetric concept of treating null values in k_i[1].

*The **`MATCH PARTIAL`** clause*

The semantics of the MATCH clause in the definition of a referential integrity constraint is according to the above definitions:

---

1. Note that the semantics of the three options are identical if n =1.

- The predicate `(tC.f1,..., tC.fn) MATCH (SELECT K FROM P)` has to be true for each tuple of C if no MATCH clause is specified.
- The predicate `(tC.f1,..., tC.fn) MATCH FULL (SELECT K FROM P)` has to be true for each tuple of C if `MATCH FULL` is specified.
- The predicate `(tC.f1,..., tC.fn) MATCH PARTIAL (SELECT K FROM P)` has to be true for each tuple of C if `MATCH PARTIAL` is specified.

The cases with no `MATCH` clause or `MATCH FULL` are covered by the discussion in the previous section. We now analyze the implications of `MATCH PARTIAL`.

The main implication of `MATCH PARTIAL` is that a child tuple may have more than one matching parent. Given a foreign key F consisting of three attributes (we will denote null values with '∅'), the foreign key of a tuple $t_C|_F$ = (x, ∅, z) will match primary keys like (x, y_1, z), (x, y_2, z) and so on. This implies that the referential actions must be refined if such a 'parent' is changed or deleted. To do so the SQL2 standard distinguishes between unique and non-unique matching parents. A tuple t_P is the unique matching parent for a tuple t_C if t_P is the only tuple in P with a primary key matching the foreign key of t_C. If t_P has a matching primary key but it is not the unique matching parent then t_P is a non-unique matching parent.

Example: Let P = {<x_1, y, z,...>, <x_2, y, z,...>} and C = {<..., x_1, y, z,...>, <..., ∅, y, z,...>}. <x_1, y, z,...> is the unique matching parent for <..., x_1, y, z,...> and a non-unique matching parent for <..., ∅, y, z,...>.

If a parent tuple t_P is deleted or updated, the referential actions are executed only for children t_C having t_P as their unique matching parent.
Given that `ON DELETE CASCADE` is defined in the example above, a deletion of <x_1, y, z,...> results in a deletion of <..., x_1, y, z,...> but not of <..., ∅, y, z,...>.
During the execution of a query in a tuple-at-a-time manner, a non-unique matching parent may become the unique matching parent (e.g. <x_2, y, z,...> for <..., ∅, y, z,...> if <x_1, y, z,...> is deleted). If this (now) unique matching parent is deleted or updated, the referential actions have to be performed. Therefore even in a single-user operation, the unique matching parent has to be evaluated dynamically. Let us now consider the different operations possibly violating referential integrity and the resulting query types if `MATCH PARTIAL` is specified. For simplicity of discussion, we concentrate on differences of the test for the match predicate and do not repeat the terms which do not change.

Delete from P

If a tuple t_P is deleted from the relation P, referential actions are only applied to child tuples t_C having t_P as their unique matching parent. Given a primary key of tuple t_P to be deleted, how can we locate the children t_C satisfying $t_C|_F$ `MATCH PARTIAL` $t_P|_K$? Since there is no MATCH predicate directly available in standard relational DBMS, we will substitute such a predicate by a number of simple predicates. A simple predicate directly supported in all DBMS is (attribute = value) and the conjunction of such terms. To find all children related via `MATCH PARTIAL` to t_P, we have to check all possible combinations of null values (with at least one defined value) in the foreign key of C; for the construction of these `MATCH PARTIAL` keys, the defined values are derived from the primary key of t_P. We will call these `MATCH PARTIAL` foreign keys F-templates. They are constructed by replacing all possible combinations of primary key values by null values. The F-template consisting of nulls only is not relevant to referential integrity.

Example: Given a tuple t_P with primary key K consisting of three attributes <x, y, z,...>, the matching foreign keys have the form <..., x, y, z,...>, <..., ∅, y, z,...>, <..., x, ∅, z,...>, <..., x, y, ∅,...>, <..., ∅, ∅, z,...>, <..., ∅, y, ∅,...>, <..., x, ∅, ∅,...>.

Apparently, for a primary key of length n we obtain $2^n - 1$ F-templates.

If the null value is represented like any other value, the overhead to select the children with matching foreign keys is a union of $2^n - 1$ queries of type (**P**, **S**)$_C$. To test whether or not referential integrity (with `MATCH PARTIAL` semantics) is violated, requires the check whether t_P is the unique matching parent of one of the matching children t_C. Two cases have to be distinguished:

- The foreign key of t_C has defined values only. Because the primary key too has defined values only, t_P is the unique matching parent and the specified referential actions are executed on t_C.
- Some attribute values in the foreign key of t_C are null. To decide whether or not referential integrity is violated, requires the location of at least one matching parent of t_C different from t_P. For example, in our three-attribute foreign key, we may have $t_C|_F$ = <x, ∅, z>. The query to locate the number of matching parents of t_C will look like

  ```
  SELECT COUNT(*)
  FROM P
  WHERE P.k1 = x AND P.k3 = z,
  ```

  which is a partial match query. We assume that t_P is already deleted and hence denote the type of this query with (**PM$_u$**, **E**)$_P$ where u denotes the number of **u**nknown attributes[1] (i.e., for the above query we have (**PM$_1$**, **E**)$_P$).

  As shown above, a primary key of length n may have $2^n - 1$ matching F-templates. Because the fully defined template is handled separately, we obtain 2^n-2 partial match queries. This set of partial match queries can be partitioned along the number of unknown values. Given a key of length n, each $1 \le u < n$ yields $\binom{n}{u}$ templates with u undefined values and therefore $\binom{n}{u}$ partial match queries of type (**PM$_u$**, **E**)$_P$. This represents the worst case, because a specific partial match query has to be evaluated only if some children exhibit the corresponding template.

Putting both results together leads to a set of queries:

Given a primary key of length n, for each referential integrity constraint referencing this primary key with `MATCH PARTIAL`, we need 2^n-1 queries of the type (**P**, **S**)$_C$ and (in the worst case) for each u, $1 \le u < n$,

1. Note that (**PM$_0$**, **E**) = (**P, E**)

a set of $\binom{n}{u}$ queries of type $(\mathbf{PM_u}, \mathbf{E})_P$. So, we can conclude $(2^n - 1) \cdot (\mathbf{P},\mathbf{S})_C + \left(\sum_{u=1}^{n-1} \binom{n}{u} \cdot (\mathbf{PM}_u,\mathbf{E})_P\right)$ as the number and the types of queries necessary to select all children having the specified parent as the unique matching parent. This set of children is subject to the referential actions.

As mentioned before, some of these queries may be optimized if the referential action `RESTRICT` is specified. In this case, the first unique matching child encountered causes the rollback of the operation. As far as referential action is concerned, the `SET DEFAULT` option represents the worst case: The existence of a "default" parent (all attributes of the foreign key are set to the default values of these attributes) has to be checked. Because null values are allowed as default values, this is a $(\mathbf{PM}_u, \mathbf{E})_P$ query. In addition, the "default" location in the access path of the children has to be selected resulting in a $(\mathbf{P}, \mathbf{S})_C$ query.

Update attribute k_i of P

Things get even worse when looking at the update of an attribute in the primary key K of P. One minor additional query concerns the check whether or not the new key is unique in P ($(\mathbf{P}, \mathbf{E})_P$). As in the case of delete, the tuple t_P has to be selected and all unique matching children have to be computed. Here, the `SET DEFAULT` option is particularly complicated. In contrast to the delete case, where the `SET DEFAULT` option causes all children to get the same default foreign key and hence the same "default" parent, in the update case for each template only the defined attributes of the foreign keys are set to the default values. Hence, a different partial match query may be necessary for each template to check whether an appropriate "default" parent exists. In case it does not exist, the complete operation is aborted. Due to this fact, the templates with the largest number of defined attributes should be checked first, because it suffices to guaranteed that one other parent exists and hence, if a parent exists for a template with m defined attributes $f_1,..., f_m$, this parent is at least matching parent for all templates composed of $f_1,..., f_m$. This observation yields that at most $\binom{n}{\lceil \frac{n}{2} \rceil}(\mathbf{PM}_u,\mathbf{E})_P$ additional partial match queries have to be evaluated. Finally the foreign key of the children has to be changed. In difference to the delete case above ($(\mathbf{PM}_u, \mathbf{E})_P + (\mathbf{P}, \mathbf{S})_C$), summarizing the maintenance costs of the children yields $\left(\binom{n}{\lceil \frac{n}{2} \rceil}(\mathbf{PM}_u,\mathbf{E})_P\right) + \langle 2^n - 1 \rangle \cdot (\mathbf{P},\mathbf{S})_C$ queries (each F-template has to be checked and updated).

Example: Given a tuple t_P with primary key K consisting of three attributes <x, y, z,...>, let t_P be the unique matching parent of the F-templates <..., x, y, z,...>, <..., ∅, y, z,...> and <..., x, ∅, z,...>. Furthermore, let us assume the default values $f_1 = a$, $f_2 = b$, $f_3 = c$. If the primary key changes from <x, y, z> to <g, h, i> then <..., x, y, z,...> is changed to <..., a, b, c,...>, <..., ∅, y, z,...> is changed to <..., ∅, b, c,...>

and <..., x, ∅, z,...> is changed to <..., a, ∅, c,...>. If <..., a, b, c,...> can be tested successfully for another parent, this parent is also an at least matching parent of <..., ∅, b, c,...> and <..., a, ∅, c,...>.

Insert into C

If a tuple t_C is inserted into the relation C, it has to be checked whether or not there is a matching parent tuple t_P. The checking query is of type ($\mathbf{PM_u}$, $\mathbf{E}$)$_P$ where u is the number of unknown attributes.

Update attribute f_j of C

If an attribute of the foreign key is updated it has to be tested whether a parent tuple for the new foreign key value exists. The overhead is the same as in the insert case: a query of type ($\mathbf{PM_u}$, $\mathbf{E}$)$_P$.

In this section, we have shown the implications of treating null values as don't-care terms while dealing with referential integrity. The main result besides the number of queries to be answered is that the queries themselves grow more complex. Without this don't-care semantics of null values, only exact match queries (type (**P**, **S**), (**P**, **T**) or (**P**, **E**)) can occur which keeps checking relatively simple. However, interpreting null values as special don't-care values changes this behavior drastically.

3.3 Overhead of Candidate Keys

So far, we have considered null values in the foreign key only. In this section, we briefly discuss the implications of null values in the referenced group of attributes K. The relational data model provides the concept of candidate keys to handle unique "identifiers" with null values: While for each tuple t_P its primary key has to be defined (no null values) and to be unique, a candidate key is a set of attributes that has to be unique only if it is fully defined. The SQL2 standard allows such candidate keys to be named as the referenced columns in the definition of a referential integrity constraint. We discuss the implications of this possibility in the following paragraph.

In the SQL2 standard, fully defined candidate keys are equivalent to primary keys (to be precise, in SQL2 a primary key is a candidate key with nulls not allowed!). But what about partially defined candidate keys being referenced in a referential integrity constraint? If the constraint does not specify any MATCH clause then a child tuple does not reference a tuple t_P of P with a partially defined candidate key because only fully defined foreign keys are considered for referential integrity. The same is true for `MATCH FULL`, where again a child tuple does not reference such a tuple because either the foreign key of the child is fully defined (referencing a fully defined candidate key) or it is completely null. Finally, if `MATCH PARTIAL` is specified, some child t_C may have a tuple t_P of P with partially defined candidate key as a matching or even the unique matching parent and, therefore, may be accessed if t_P is updated or deleted. Hence, only if `MATCH PARTIAL` is specified, candidate keys are of interest in the scope of this paper.

We are interested in the query types to be supported for efficient referential integrity checking. Due to the (asymmetric) definition of the `MATCH` predicate (if defined symmetrically, it would be more like a unification than a matching) all matching foreign keys for a partially defined candidate key are null at least in those attributes where the referenced attribute is null, as well. The other attributes (not null in the foreign key) are handled the same way as before. Therefore, there is no change in the query types, only the number of queries may decrease.

Example: Given a candidate key of length n and a tuple having v undefined attributes, then the n-v attributes form the new “key” and the formulas above apply for this number. In the worst case, this yields up to $(2^{n-v}-1)\cdot(\mathbf{P},\mathbf{S})_C + \left(\sum_{u=v}^{n-1}\binom{n}{u}\cdot(\mathbf{PM}_u,\mathbf{E})_P\right)$ queries for selecting the unique matching children.

Because handling of candidate keys does not embody new aspects, in the following we will assume that the referenced attributes in a referential integrity constraint constitute the primary key of the referenced relation.

3.4 Summary

So far, the purpose of our discussion was to introduce the specification of referential integrity in SQL2 and to deduce from this specification the functional requirements for query processing in order to maintain referential integrity. Based on the referential action `SET DEFAULT`, the following table summarizes the types and the number of queries needed to select the tuples for checking and enforcing referential integrity (through referential actions).

Table 1 shows the query requirements of `SET DEFAULT` which represents the most complex case among the referential actions. Since we don’t know which of the referential actions are preferred in real world applications, we summarize the differences concerning `RESTRICT` case as the most simple referential action in Table 2. The corresponding figures can be derived from Table 1 by removing the terms

	no MATCH clause or `MATCH FULL`	`MATCH PARTIAL`
Insert into P	$(\mathbf{P},\mathbf{E})_P$	$(\mathbf{P},\mathbf{E})_P$
Delete from P	$(\mathbf{P},\mathbf{T})_P + (\mathbf{P},\mathbf{E})_P + 2\cdot(\mathbf{P},\mathbf{S})_C$	$(\mathbf{P},\mathbf{T})_P + \left((2^n-1)\cdot(\mathbf{P},\mathbf{S})_C + \overbrace{\left(\sum_{u=1}^{n-1}\binom{n}{u}\cdot(\mathbf{PM}_u,\mathbf{E})_P\right)}^{=\,2^n-2\text{ queries}}\right) + (\mathbf{PM}_u,\mathbf{E})_P + (\mathbf{P},\mathbf{S})_C$

Table 1: Summary of the query requirements for supporting referential integrity (`SET DEFAULT`)

	no MATCH clause or `MATCH FULL`	`MATCH PARTIAL`
Update attribute k_i of P	$(\mathbf{P}, \mathbf{T})_P + 2 \cdot (\mathbf{P}, \mathbf{E})_P + 2 \cdot (\mathbf{P}, \mathbf{S})_C)$	$(\mathbf{P},\mathbf{T})_P + \left((2^n - 1) \cdot (\mathbf{P},\mathbf{S})_C + \overbrace{\left(\sum_{u=1}^{n-1} \binom{n}{u} \cdot (\mathbf{PM}_u,\mathbf{E})_P\right)}^{= 2^n - 2 \text{ queries}}\right) + (\mathbf{P}, \mathbf{E})_P + \left(\left(\left(\binom{n}{\lceil \frac{n}{2} \rceil}\right)(\mathbf{PM}_u,\mathbf{E})_P\right) + \langle 2^n - 1 \rangle \cdot (\mathbf{P},\mathbf{S})_C\right)$
Insert into C	$(\mathbf{P}, \mathbf{E})_P + (\mathbf{P}, \mathbf{T})_C$	$(\mathbf{PM}_\mathbf{u}, \mathbf{E})_P + (\mathbf{P}, \mathbf{T})_C$
Delete from C	$(\mathbf{P}, \mathbf{T})_C$	$(\mathbf{P}, \mathbf{T})_C$
Update attribute f_j of C	$(\mathbf{P}, \mathbf{E})_P + 2 \cdot (\mathbf{P}, \mathbf{T})_C$	$(\mathbf{PM}_\mathbf{u}, \mathbf{E})_P + 2 \cdot (\mathbf{P}, \mathbf{T})_C$

Table 1: Summary of the query requirements for supporting referential integrity (`SET DEFAULT`)

which represent the update of the children and by changing the search for all children ($(\mathbf{P}, \mathbf{S})_C$) into a lookup of one child given a specific foreign key ($(\mathbf{P}, \mathbf{E})_C$).

	no MATCH clause or `MATCH FULL`	`MATCH PARTIAL`
Delete from P	$(\mathbf{P}, \mathbf{T})_P + (\mathbf{P}, \mathbf{E})_C$	$(\mathbf{P},\mathbf{T})_P + \left((2^n - 1) \cdot (\mathbf{P},\mathbf{E})_C + \left(\sum_{u=1}^{n-1} \binom{n}{u} \cdot (\mathbf{PM}_u,\mathbf{E})_P\right)\right)$
Update attribute k_i of P	$(\mathbf{P}, \mathbf{T})_P + (\mathbf{P}, \mathbf{E})_P + (\mathbf{P}, \mathbf{E})_C$	$(\mathbf{P},\mathbf{T})_P + \left((2^n - 1) \cdot (\mathbf{P},\mathbf{E})_C + \left(\sum_{u=1}^{n-1} \binom{n}{u} \cdot (\mathbf{PM}_u,\mathbf{E})_P\right)\right) + (\mathbf{P}, \mathbf{E})_P$

Table 2: Summary of the query requirements for supporting referential integrity (`RESTRICT`)

4 Access Path Support for Referential Integrity Checking

So far, the discussion of the update and maintenance operations in the parent and child relations has revealed the typical search operations necessary to locate the tuples involved in checking key uniqueness and referential integrity. Insertion of a new parent requires one or more UNIQUE conditions to be checked (for primary key and each candidate key). When a parent is deleted or its primary key is updated, the set of related children has to be located via their foreign key to perform the specified referential actions which themselves may demand primary or foreign key access. Insertion of a new child tuple requires the examination of multiple key conditions (primary, candidate, and foreign keys). Furthermore, the modification of a foreign key in a child tuple implies the checking of whether or not a parent exists with a primary key value equal to the new foreign key value. (Subsequently, we do not consider candidate keys; their search and maintenance cost may be estimated from the primary key).

In all these situations, the absence of appropriate access paths would enforce the use of sometimes multiple sequential scans to perform uniqueness tests, existence tests, or the search of the parent and child tuples related by the referential integrity constraint. Parallelism does not seem a panacea to cope with these sequential scans. For large relations, only massive parallel architectures would provide the required speed-up; such an approach, however, introduces severe I/O and partitioning costs. As a consequence, the response time degradation caused by searches in sufficiently large relations is not tolerable for most applications. Therefore, DBMSs must allocate index structures for all types of keys to efficiently maintain all relational invariants. In our case, we only focus on the keys K and F and the referential integrity defined between them to derive the operational search cost of referential integrity maintenance.

Hence, suitable access paths for (**P**, **T**), (**P**, **S**) and (**P**, **E**) as well as for ($\mathbf{PM_u}$, **E**) have to be provided to determine the uniqueness of a primary key and the matching predicates of the primary key and the foreign key, thereby speeding up the search process. On the other hand, these access paths cause additional overhead whenever an operation modifies the set of existing K- or F-key values (shown as additional terms in the requirements analyses). To allow the comparison of search costs we introduce the number of logical page references (or page references for short) C_R needed to traverse the access path data in order to perform the requested task. Since C_R is independent of the run-time environment, it should facilitate a comparison of using different access paths. We assume that the cost measure C_R for all access-path-related searches needed for an update operation is indicative for the overall costs including the access-path-related update and logging as well as the required locking of the search paths to guarantee repeatability of reads (consistency level 3 [Gr78]). Including these additional costs would require a much more detailed modelling of access paths, operations, and multi-user environment [Mo90]. Since we wish to determine only the order of the overhead related to referential integrity and to compare the usefulness of different access paths, our simplification seems to be justified.

As shown in Sect. 3, certain search operations ((**P**, **T**), (**P**, **S**) and (**P**, **E**)) on attributes K and F may be anticipated very often. As a consequence, index structures for relations P and C, denoted by $I_P(K)$ or $I_C(F)$, have to be available supporting the following operations:

- direct search for a key value in the index structure for checking the UNIQUE option, for finding the record address, and for the insertion/deletion of entries.
- successive access to all keys having the same value or belonging to a given key range.
- direct search for a foreign key and the corresponding primary key or for a primary key and the related set of foreign key values.

4.1 Support for the Regular MATCH Options

Because of their frequency, these search operations are very important for the overall performance of a DBMS and for this reason, they have to be supported sufficiently well. Let us first focus on the case where n=1 for K and F. Then, a standard candidate for implementing an index structure is the B*-tree.

B*-tree

The structure of a B*-tree [Co79] representing for example an index for attribute DNO in relation DEPT is illustrated by Fig. 1. The corresponding leaf page illustrates the format of a UNIQUE index. In addition, the leaf-page format of a NONUNIQUE index containing TID-lists is shown. Since the key values and the related TIDs or TID-lists are allocated in a key-sorted sequence to the leaf pages and since these pages are linked together with NEXT and PRIOR pointers, successive access to all key values or to a given key range is performed efficiently.

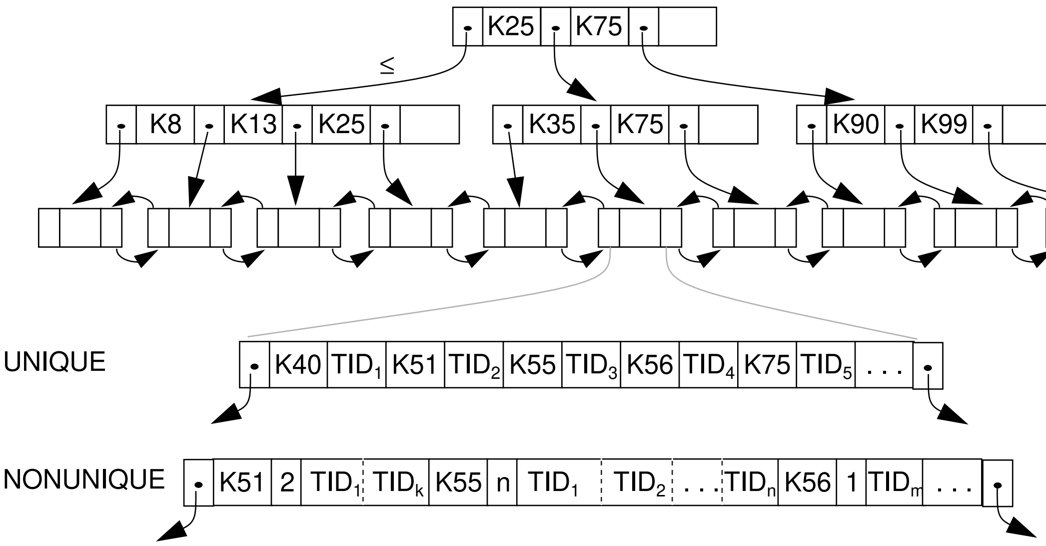


Fig. 1: I_{DEPT}(DNO) as a B*-tree

According to its definition, a page (except the root) of a B*-tree has at least k > 0 and at most 2k key/ TID-pairs called index entries. Hence, the height is delimited by the following formulas:

$$(4.1) \qquad h \geq 1 + \log_{(2k+1)}\left(\frac{N}{2k}\right) \quad \text{and (4.2)} \qquad h \leq 2 + \log_{(k+1)}\left(\frac{N}{2k}\right)$$

where N>0 is the number of indexed tuples.

The access costs to locate a key in the B*-tree are C_R = h page references. The expensive fraction of the overall cost is the number of physical I/O operations $C_{I/O}$ required to perform the tree traversal. Depending on the locality of reference on such B*-trees, the size of the database buffer, the replacement algorithm etc. $C_{I/O}$ may be less than C_R, because some pages in the path to be traversed in the B*-tree may be already residing in the buffer thereby saving physical I/Os to the disk.

According to Table 1, searches for referential integrity maintenance are dependent on the type of operation. If both index structures $I_P(K)$ and $I_C(F)$ are implemented by separate B*-trees, the specific operations may be sketched as follows. “Insert into P” and “Delete from C” are very simple, as far as our checking task is concerned. Accessing $I_P(K)$ and $I_C(F)$ for checking the uniqueness of the K-key value and for removal of the F-key value needs a tree traversal of h_K and h_F page references, respectively. “Delete from P” with the referential action `SET DEFAULT` (worst case) requires a traversal of $I_P(K)$ to locate the primary key to be deleted and two traversals to the location of the foreign key and the DEFAULT key in $I_C(F)$. If a DEFAULT key does not exist in $I_C(F)$, it will be inserted. Furthermore, $I_P(K)$ will be accessed to make sure that a DEFAULT key is in P. Hence, the corresponding number of page references may sum up to $C_R = 2h_K + 2h_F$. Furthermore, “Insert into C” causes the insertion of an F-key value and a check whether or not the related K-key value is present. Hence, $C_R = h_K + h_F$. Eventually, both update operations in P and C may be composed of delete and insert operations, as far as tree traversal is concerned.

To enable a simple comparison, a synopsis of all cost formulas is contained in Table 3.

Combined access path

Since the keys K and F are defined on the same domain, it is possible to implement both index structures by a common B*-tree called combined access path structure (CAPS for short) in [Hä78]. Because of the given operational characteristics, it seems to be advantageous to combine the index structures $I_P(K)$ and $I_C(F)$ to reduce I/O. The non-leaf pages of the B*-tree contain a unified reference structure for both indexes. In the leaf pages, the entries for $I_P(K)$ and $I_C(F)$ are combined according to the following format (here showing a UNIQUE and a NONUNIQUE option). In this example, K and F map to domain D with value D_i:

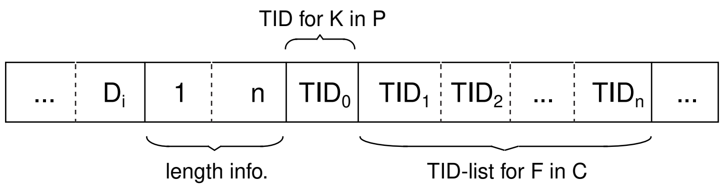


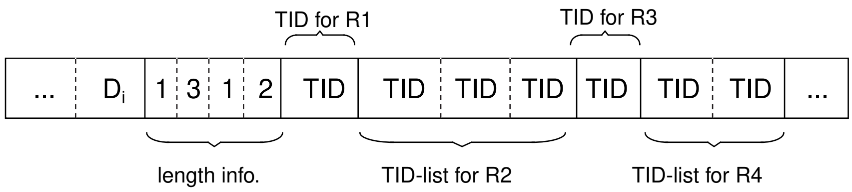
Fig. 2: Leaf-page format for a CAPS

As compared to a single-index B*-tree for $I_P(K)$ and h_K, the height h_{KF} of the resulting B*-tree is typically not changed because the horizontal growth is dominant in B*-tree structures. Only in rare cases, an increment of the height by one of has to be anticipated ($h_{KF} \leq h_K+1$, see Fig. 4). Since subsequent access to corresponding key values of K and F uses the same tree traversal, locality of reference is further improved. As a consequence, the CAPS offers salient features for checking referential integrity as well as a substantial cost reduction as compared to separate B*-trees.

Both indexes $I_P(K)$ and $I_C(F)$ supporting $(\mathbf{P}, \mathbf{T})_P$ and $(\mathbf{P}, \mathbf{S})_C$ are mapped to a single B*-tree using their domain values. Since in various situations the same tree traversal can be used to locate the F-key and

K-Key values, several page references can be saved. For example, “Delete from P” with the referential action `SET DEFAULT` now requires only $C_R \leq 2h_{KF}$ page references to locate the key to be deleted and the DEFAULT key in the CAPS. The cost figures for the remaining operations may be derived in a similar way. They are summarized in Table 3.

The idea sketched in Fig. 2 may be used to support more pairs of referential relationships; it can be applied to the situation where m relations with j candidate and foreign keys ($m \leq j$) defined on the same domain have to be indexed. Such structures are called generalized access paths in [Hä78]. For example, the format of the leaf page is illustrated in the following for m = j = 4:



Of course, the benefit of such access paths for checking referential integrity is increasing with the number j of the keys involved.

Join index

A similar structure to the combined access path was proposed in [Va87] as a so-called join index, which primarily aims at the optimization of the 2-way join. It is defined for two relations P and C as follows:

$$JI = \{(TID_P, TID_C) \mid f(p.A, c.B) \text{ is TRUE}, p \in P, c \in C\}.$$

f denotes an arbitary join predicate. Apparently, JI may serve to embody materialized Θ-joins by surrogates or TIDs. If an equi-join is used and the primary and foreign keys K and F are chosen for A and B, then the parent and child TIDs with matching K- and F-key values are stored together by a join index. At first sight, this information could be useful to support referential integrity checking. However, this purpose is complicated since a join index does not use key values but only TIDs. Moreover, the direct representation of the join index JI as a binary table does not provide access support (other than sequential) for TID_P as well as for TID_C.

In our evaluation context, we assume that a join index is specified for an equi-join combining primary key of P and foreign key of C in a (1:n)-manner. Furthermore, symmetric and fast access is needed to perform efficient operations on the joined view. Hence, the logical JI table has to be implemented as two clustered index structures [Va87], i.e., sorted according to TID_P and TID_C. As a consequence, we obtain the indexes $I_{PC}(TID_P)$ and $I_{CP}(TID_C)$ with heights h_{PC} and h_{CP}, respectively.

As illustrated by Fig. 3a, these index structures do not permit access by primary or foreign key values. To use these structures for referential integrity maintenance and for other kinds of search requests, additional index structures are necessary to map the key values to their related TIDs. Fig. 3b shows $I_P(K)$

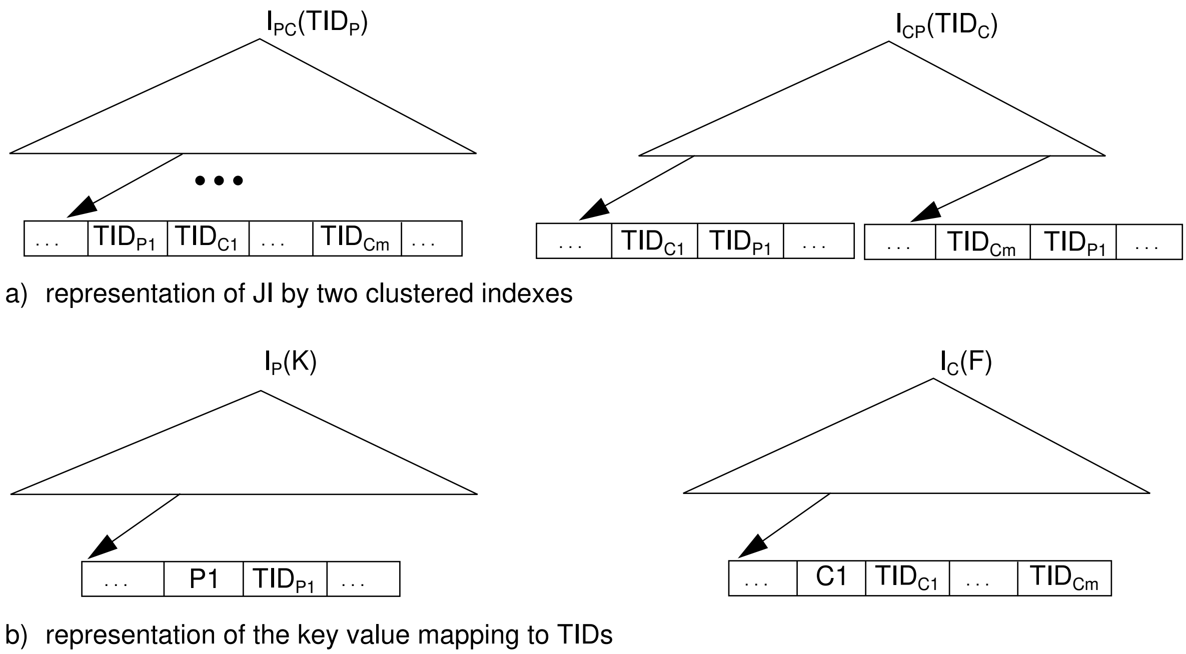


Fig. 3: Mapping of a join index to a set of B*-trees

and $I_C(F)$ which are identical to the indexes used in the pure B*-tree approach. Looking at Fig. 3, it becomes immediately obvious that the join index does not speed-up the access behavior to check referential integrity constraints since $I_{PC}(TID_P)$ and $I_{CP}(TID_C)$ are redundant as far as referential integrity is concerned.

To compare this solution with the pure B*-tree and the CAPS, we have listed the search costs for the update operations on P and C in terms of page references in Table 3. In this case, the evaluation of (**P**, **T**)$_P$ and (**P**, **S**)$_C$ has to be mapped to the four B*-trees of Fig. 3. In our discussion, we only sketch some operation and leave the cost modeling of the remaining operations to the reader. Our cost formulas are listed in Table 3.

The worst effect on search costs has "Delete from P" with the referential action `SET DEFAULT`. In a first step, the location of the K-key value, e.g. P1, to be deleted has to be identified in $I_P(K)$ delivering TID_{P1} which is, in turn, used to search (TID_{C1},..., TID_{Cm}) via $I_{PC}(TID_P)$. So far, we have accomplished $h_K + h_{PC}$ page references. Note the tuples t_{C1} are not deleted but allocated to a parent t_{Pdef} incorporating the DEFAULT key. For this reason, we have to assure the existence of that key via $I_P(K)$ (h_K) and to move the set of (TID_{C1},..., TID_{Cm}) to the corresponding entry TID_{DEF} of t_{Pdef} in $I_{PC}(TID_P)$ (h_{PC}).

In $I_C(F)$, the location of the matching foreign key value (C1) and the location of the DEFAULT value have to be found in order to delete the key entry and to move the corresponding list of TIDs. Hence, we additionally obtain $2h_F$ page references. Finally, we have to copy the TID_{DEF} to m entries in $I_{CP}(TID_C)$ which requires $m \cdot h_{CP}$ page references in the worst case.

Comparison of access path structures

Table 3 compares the search costs for referential integrity maintenance when different index implementations are used. Apparently, the join index is not appropriate at all. This structure guarantees fast and symmetric access clustered by surrogate values to the entire joined relations. These access characteristics, however, have to be maintained when both base relations are modified. As a consequence, update operations referring to elements involved in referential integrity checking automatically lead to a modification of the materialized join structure. The indirection of key to TID incorporates an additional penalty for this structure.

As already discussed, the CAPS not only supports two indexes on one B*-tree, it further accomplishes the joining and checking of the related K-key and F-key values for free. In addition to the C_R values shown, due to the much better locality of reference the "real performance" measured in physical I/Os is even superior for the CAPS as compared to the B*-tree. The values given in Table 3 are derived for the referential action `SET DEFAULT`. The support of the `RESTRICT` option does not change the cost formulas dramatically for the B*-tree and CAPS solutions, e.g., "Delete from P" yields $h_K + h_F$ resp. h_{KF}.

	2 B*-trees	CAPS	Join index
Insert into P	h_K	h_{KF}	h_K
Delete from P	$2 \cdot h_K + 2 \cdot h_F$	$2 \cdot h_{KF}$	$2 \cdot (h_K + h_F) + 2 \cdot h_{PC} + m \cdot h_{CP}$
Update attribute k_i of P	$3 \cdot h_K + 2 \cdot h_F$	$3 \cdot h_{KF}$	$3 \cdot h_K + 2 \cdot h_{PC} + 2 \cdot h_F + m \cdot h_{CP}$
Insert into C	$h_K + h_F$	h_{KF}	$h_K + h_{CP} + h_F + h_{PC}$
Delete from C	h_F	h_{KF}	$h_K + h_{CP} + h_{PC}$
Update attribute f_i of C	$h_K + 2 \cdot h_F$	$2 \cdot h_{KF}$	$2 \cdot h_K + 2 \cdot h_{CP} + h_K + 2 \cdot h_{PC}$

Table 3: Summary of the results (page references) for `MATCH FULL` and missing MATCH clause

A problem complicating the interpretation of Table 3 are the various cost parameters. To relate the various heights, let us consider the critical factors which determine the height of a B*-tree, namely the number of tuples to be indexed (N) and the number of index entries (TID/key-pairs) per page. The latter is dependent on the page size itself and the average length of an entry (e). Obviously, e critically determines the fan-out of the tree. With a TID-length of 5 bytes, 4 bytes as the page pointer and 1 byte as an offset, we can access 2^{32} pages and 256 tuples within a page which are reasonable numbers. In contrast, the sizes of the keys may vary over a considerable range, e.g. an employee number needs 4 bytes, whereas a name may require 40 bytes or more. To improve fan-out in such cases, key compression may be used successfully, i.e., as reported in [NMR79, Wa73], front and rear compression resulted in an average length for compressed keys of 1.78 bytes (+ 2 bytes of organizational data) having originally 20-byte keys.

In order to derive stable estimations for the various heights, we attempt to express the sensitivity of height changes depending on N and e. The range in which a given value of h occurs is delimited by the two situations where each node of the B*-tree has a minimum resp. maximum number of index entries, i.e., k (except for the root) resp. 2k entries. These delimiting conditions are characterized by h_{Nmin} and h_{Nmax}. As indicated by the formulas 4.1 and 4.2, these values are determined by N_{min} resp. N_{max} and k which, in turn, is given by the page size p and the average entry length e, i.e., for a given p, h_{Nmin} and h_{Nmax} are functions in N and e. Fig. 4 illustrates the isolines for various values of h_{Nmin} and h_{Nmax} within

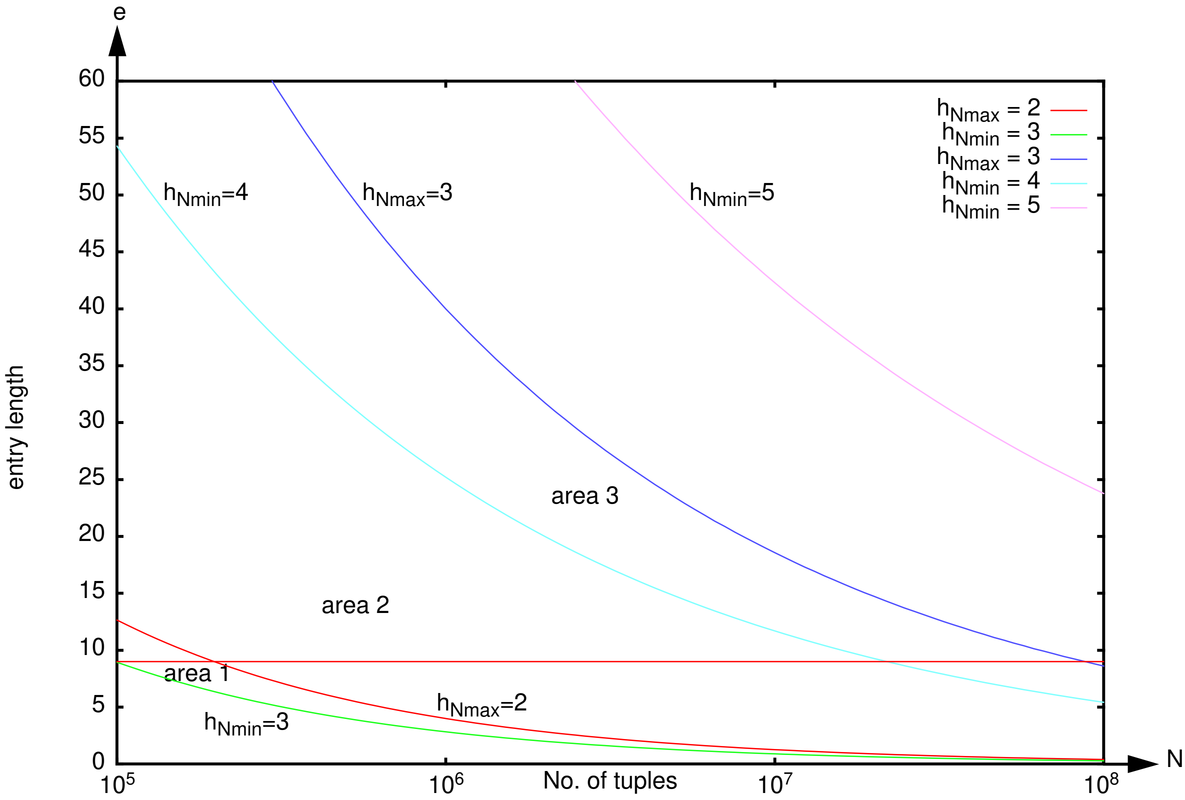


Fig. 4: Relating N, e and h of a B*-tree

practical ranges for N and e given a page size of 4K bytes. For example, the area between the isolines of h_{Nmin} = 3 and h_{Nmax} = 3 is further divided into 3 subareas by h_{Nmax} = 2 and h_{Nmin} = 4. Area 1 represents (N/e)-pairs with h = 3 or better whereas h = 3 results from all (N/e)-pairs in area 2. Finally, (N/e)-pairs in area 3 may obtain heights of 3 or 4.

We assume a minimum index entry length e of 9 bytes which results from a TID-length of 5 bytes and the use of key compression [Wa73]. As shown in Fig. 4, for e = 9 a difference of two orders of magnitude in N may yield B*-trees of the same height h, or in an increase to h + 1, at the most. For example, when the parent and the child relations contain 10^6 resp. up to 10^8 tuples, $h_K = h_F$ (=3) or $h_F = h_K + 1$ (=4). The same observation is true for other practical values of e. Note, the use of TID-lists for multiple references of the same key value saves additional space in the leaf pages of a B*-tree, especially for larger values of e, thus keeping h constant for even larger ranges of N. Furthermore, increasing the page size (e. g.

to p = 8K) will dramatically increase the fan-out thereby making the height of a B*-tree much more insensitive to the growth of N. These considerations justify the following approximation:

- For symbolic manipulations of the cost formulas, we assume $h_F = h_K = h_{KF} = h_{CP} = h_{PC}$ or $h_F = h_{KF} = h_{CP} = h_{PC} = h_K + 1$. As a result of this approximation, we get cost formulas depending on one parameter h. The variance of the height is denoted by [+1].
- To achieve indicative numbers we will use h = 3, assuming a scenario as depicted in Fig. 4.

	2 B*-trees	CAPS	Join index
Insert into P	h 3	h [+ 1] 3 -- 4	h 3
Delete from P	4 · h [+ 2] 12 -- 14	2 · h [+ 2] 6 -- 9	h · (6 + m) [+ m + 4] (18 + 3·m) -- (22 + 4·m)
Update attribute k_i of P	5 · h [+ 2] 15 -- 17	3 · h [+ 3] 9 --12	h · (7 + m) [+ m + 4] (21 + 3·m) -- (25 + 4·m)
Insert into C	2 · h [+1] 6 -- 7	h [+ 1] 3 -- 4	4 · h [+ 3] 12 -- 15
Delete from C	h [+1] 3 -- 4	h [+ 1] 3 -- 4	3 · h [+ 2] 9 -- 11
Update attribute f_i of C	3 · h [+ 2] 9 -- 11	2 · h [+ 2] 6 -- 8	7 · h [+ 4] 21 -- 25

Table 4: Summary of the approximated results

Apparently, Table 4 clearly illustrates the advantages of the CAPS solution: cost-effective access and insensitivity to growth of the underlying relation. For these reasons, it is the superior alternative to support the regular MATCH option.

So far, we have discussed access path solutions for operations requiring only point queries (**P**, **T**) and (**P**, **S**). Although the examples were shown for n = 1, B*-trees and CAPSs allow simple extensions to larger n. In these cases, the n values of a key are encoded as a compound- key value [BCE77] such that the point queries can be supported easily. As a result, the relative cost figures remain stable whereas the heights of the various trees may change slightly (see Fig. 4).

4.2 Support of the `MATCH PARTIAL` option

Maintenance of the referential integrity becomes much more complicated when the `MATCH PARTIAL` option is used, since then partial match queries in addition to point queries are required.

Two types of operation are discussed in some detail to cover the requirements of access path support: “Insert into C” and “Delete from P”. For the remaining operations, the analysis of search costs for referential integrity maintenance is left to the reader. Update of C is just the combination of the delete and insert operations, whereas update of P is much more complicated because of the non-symmetrical semantics of the referential actions (see Sect. 3.2).

We assume that K and F consist of n attributes (n > 1) and that u attributes in F (u < n) may be undefined; K is a primary key and all its attributes have defined values[1]. Furthermore we concentrate the discussion on the costs for the support of `MATCH PARTIAL` with the option `RESTRICT`, i.e., the costs for selecting the parent t_P and those children with t_P as their unique matching parent (thereby disregarding further update overhead provoked by other referential actions which may even double the cost in case of `SET DEFAULT`). For this reason, we often refer to partially defined foreign keys, e. g. <x, ∅, z>, using

- a point query to determine whether a corresponding tuple exists in C, and
- a partial match query to check whether matching parents exist in P; these are found by applying the related search key <x, –, z> where '–' denotes the don't-care value.

General aspects of the evaluation

Before we enter the discussion of access support for the `MATCH PARTIAL` option, we will recall the most important steps of our reference operations in a more abstract way.

Insert into C

To check referential integrity when inserting a tuple t_C is simple as long as the F-key value is fully defined (e.g., <x, y, z>) or fully undefined (e.g., <∅, ∅, ∅>). The former case is handled by a point query to the P relation, whereas the latter case does not need a check. All other templates of the F-key are more difficult and imply a partial match query to identify matching parents. As soon as the first matching parent is found, the check condition is satisfied and the tuple t_C can be inserted.

Delete from P

The deletion operation locates the parent tuple t_P with primary key <x, y, z>. If it exists, it is deleted which may cause a violation of the `PARTIAL MATCH` semantics of referencing tuples in C. All unique matching children of t_P have to be determined in C to apply the specified referential actions.

Obviously, all children with foreign key <x, y, z> match uniquely. In addition to this 'full match' relationship, tuples with partially matching foreign keys may be affected by the deletion of t_P. A child tuple may have more than one parent tuple and vice versa (n:m) which introduces substantial complications. All children having F-keys partially defined w.r.t. <x, y, z> may match either t_P uniquely or match multiple parents. Hence, in order to decide whether referential actions have to be applied, we have to inspect whether besides t_P some other parent exists. For this reason, all partially defined F-keys have to be investigated. Roughly, two different approaches are conceivable. A **straight-forward method** would proceed as follows: In a first step, the potentially affected tuples in C are determined by 2^n - 1 point queries using all possible templates for the F-key. Each successful query requires a check for a matching parent.

1. We will use n=3 for illustration purposes.

This can be decided directly for the fully defined F-key whereas other templates have to be transformed to search keys for partial match (by replacing '∅' by '–') to check for matching parents in P. In a second step, m_1 ($0 \le m_1 \le 2^n - 2$) partial match queries are evaluated in P (a single hit suffice). All unsuccessful queries indicate the templates and, in turn, the tuples in C for which referential actions have to be applied.

Apparently, this procedure is very expensive requiring $2^n - 1$ point queries and in the worst case $2^n - 2$ partial match queries. Moreover, since multiple tuples in P may match a given template, it may happen that matching tuples in P exist for all templates. In this case, further referential actions are avoided. To exploit such anticipated situations, we propose the following **inverse check procedure**: All applicable $2^n - 2$ templates are generated from <x, y, z>, transformed to resp. search keys, and executed as partial match queries on P. Each of the m_2 ($0 \le m_2 \le 2^n - 2$) unsuccessful queries requires a point query to C to determine whether tuples exist for the corresponding templates. In addition, C has to be accessed for the fully defined F-key. Hence, 1 up to $2^n - 1$ point queries to C may result.

In both approaches, the evaluation of up to $2^n - 2$ partial match queries in P is a key factor of the overall costs. Their sequential evaluation would introduce a considerable share of redundant processing, since some queries are not independent from each other. For n = 3, assume the queries

q_1: x, y, –

q_2: x, –, z

q_3: x, –, –

Then, q_1 and q_2 are special cases of q_3, or with other words, if we evaluate q_3, we can use the derived set of keys to check whether q_1 and q_2 can be satisfied. Hence, we only have to determine the key sets qualified by all partial match queries having $u = n-1$ don't-care values. Note, that these are only n queries; however, they are the most expensive among the q_i ($i \le 2^n - 2$), because they show the least selectivity. If we buffer the resulting key lists, we can answer the $2^n - 2$ partial match queries by processing only the n queries having a single key component defined. Of course, this search optimization has to be adjusted to the characteristics of the access paths used.

The inverse check procedure has to evaluate all possible partial match queries, since all templates are generated disregarding the actual tuples in C. However, the search optimization will save a lot of effort. On the other hand, the straight-forward procedure has to perform $2^n - 1$ point queries. Fig. 5 shows the different evaluation paths of the procedures. The question which procedure is superior depends on the relation between following cost functions:

1. The straight-forward method (①)results in $(2^n - 1) \cdot (P, S)_C + m_1 \cdot (PM_u, E)_P$, where m_1 represents the number of templates found in C which have to be checked for parents.

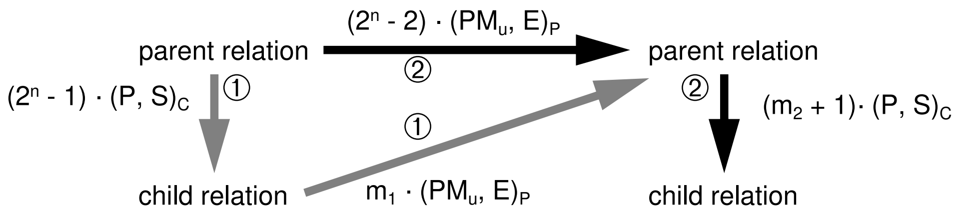


Fig. 5: Evaluation alternatives for `MATCH PARTIAL` support

2. The alternative method yields (②) $(2^n - 2) \cdot (PM_u, E)_P + (m_2+1) \cdot (P, S)_C$, where m_2 represents the number of templates for which no parents were found in P and, therefore, possibly existing children have to be located to carry out the referential action.

In the worst case, $m_1 = m_2 = 2^n - 2$ and hence, ① and ② are equally expensive. An exact analysis is hardly possible since data distribution and usage of null values within the foreign keys have to be known. In real applications, we expect very small numbers for m_1 and m_2. Due to space limitations, we restrict ourselves on the further inspection of ② and use $m = m_2$ within the cost formulas.

Use of compound-key B*-trees

Two compound-key B*-trees are used to implement $I_P(K)$ and $I_C(F)$. In both trees, the n attribute values belonging to a key are concatenated and encoded as a single field [BCE77]. Note, since null values are treated as special values, keys (e.g. <x, ∅, z>) can be represented as regular values in the B*-tree. Hence, F-keys with null values are stored in an encoded form, too. Key comparison is achieved by special encoding procedures.

The most difficult action in the "Insert into C" operation is the partial match search in P. How can we perform such a search, if search keys like <x, –, z> or <–, y, –> are given? In such cases, a search on all fully defined K-key values of $I_P(K)$ has to be accomplished to determine matching parents according to the `MATCH PARTIAL` semantics.

Note that scanning all compound keys cannot be avoided, since entering the B*-tree using partially defined search keys is hardly possible. A specialized search procedure based on some kind of prefix comparison could be designed only for the case where the first search key components are defined. Therefore in other cases, a reasonable search procedure would be a leaf page scan on $I_P(K)$ starting from the leftmost leaf to the rightmost leaf. Each encoded K-key is compared with the search key until a valid substitution is found for the partial match predicate. Hence, the access overhead is limited to $C_R = h_K + N_{Kleaf} - 1$ (N_{Kleaf} = # of leaf pages). Furthermore, the insertion point in $I_C(F)$ has to be located (h_F).

"Delete from P" comprises the deletion of t_P and the corresponding children having fully matching foreign keys. Locating the deletion point in $I_P(K)$ requires h_K page references. According to the inverse check procedure, 2^n - 2 partial match queries have to be evaluated on P. With the compound-key B*-tree for $I_P(K)$, each partial match query can be effectively executed by a leaf page scan. Each scan can be finished as soon as a valid substitution of the F-key is found in the K-key. Hence, the worst case overhead of page references is

$$C_R = (2^n - 2) \cdot (h_K + N_{Kleaf} - 1).$$

As indicated above, the number of queries can be reduced to n when the list of qualifying K-keys can be cached in main memory for further tests. Moreover, it should be possible to design an optimized search and check procedure performing a single leaf page scan which searches for all n keys thereby limiting the worst case overhead of page references to

$$C_R = (h_K + N_{Kleaf} - 1).$$

Depending on the outcome of these tests, an inspection of C is necessary to find out whether certain F-keys (templates) exist or not. Hence, m + 1 point queries to C have to be taken into account. Using $I_C(F)$ for their evaluation, the sum of page references is $C_R = (m + 1) \cdot h_F$.

Obviously, the compound-key solution could be implemented by a CAPS with the concatenated key values representing an artificial domain. Although the height of the resulting B*-tree is similar to h_F, the number of leaf pages may be much more than doubled as compared to N_{Kleaf} due to the added foreign key entries of relation C. Since N_{Kleaf} is already a very large factor, most operations would deteriorate drastically (see Table 5). Thus, the solution based on a CAPS is not favorable for such a use of compound keys.

Use of simple-key B-trees*

Apparently, compound-key B*-trees are inappropriate for partial match search. To explore a better solution we propose an opposite approach by representing the n attributes of K (or F) by n single-key B*-trees for all k_i of K and f_i of F. Since all attributes may be accessed separately or in combination, a much greater flexibility for query processing may be achieved. However, referential integrity checking seems to become more complicated. Since the reference information is distributed across multiple B*-trees, the basic checking mechanism is to fetch the qualifying TID-lists for k_i or f_i values and to merge them in order to identify the TIDs of the parents or children.

Insertion of a tuple t_C has to determine the existence of a matching tuple t_P. As explained in Sect. 3.2, this task can be accomplished by finding at least one tuple t_P whose key is a valid substitution for the newly inserted F-key of the tuple t_C. Assume n-u ($0 \leq u < n$) attributes f_i have a defined value (e.g., f = <x, ∅, z>). Then, all defined values (excluding null) are used for the search in the B*-trees for the corresponding attributes k_i. Each of these (n-u) TID-lists (e.g., $L(k_1 = x)$) with length l_{ki}, $1 \leq i \leq n-u$ is brought to main memory for an existence test of some tuple t_P (for simplicity let the first n-u attributes of the foreign key be those with defined values). If the intersection $L(k_1 = x) \cap ... \cap L(k_{n-u} = z)$ is not empty, a tuple

t_P exists whose key K coincides in the defined values with the F-key values. Since the remaining K-key values are defined and since any key value is a valid substitution for a null value, the identified K-key values, in turn, satisfy the integrity constraint. Obviously, the number of page references is

$$C_R = \sum_{i=1}^{n-u}\left(h_{ki} + \frac{l_{ki} \cdot e}{p}\right)$$, where e is the length of a TID and p is the page size.

In the following, we approximate h_{ki}, l_{ki} resp. h_{fi}, l_{fi} by the corresponding average values for the heights and list lengths, i.e., $C_R = (n-u) \cdot \left(h_K + \left(\frac{l_K \cdot e}{p}\right)\right)$. Furthermore, locating the n insertion points for the foreign key of t_C requires n single-key B*-tree traversals and the manipulation of the resulting lists which sums up to $n \cdot \left(h_F + \left(\frac{l_F \cdot e}{p}\right)\right)$. Hence, the entire overhead is

$$C_R = n \cdot \left(h_F + \left(\frac{l_F \cdot e}{p}\right)\right) + (n-u) \cdot \left(h_K + \left(\frac{l_K \cdot e}{p}\right)\right)$$ page references.

Deletion of a tuple t_P is more complicated. After the n deletion points in the B*-trees of the K-key are located causing $n \cdot \left(h_K + \left(\frac{l_K \cdot e}{p}\right)\right)$ pages referenced to propagate the deletion of t_P, in a next step, $2^n - 2$ partial match queries are to be evaluated in P. Therefore, the following test is carried out for each search key: Let u be the number of don't-care values and (for simplicity) k_i, $1 \le i \le n-u$, the defined attributes, e.g., the search key has the form <x,..., z, –,..., –>. If the intersection of the already selected lists $L(k_1 = x) \cap ... \cap L(k_{n-u} = z)$ is not empty, then there is at least one matching parent. For our evaluation, we anticipate m queries having empty intersections. Consequently, for the related templates and in addition for the fully matching foreign key, we have to check whether or not any children exist. This step requires m+1 point queries on C which can be executed by subsequently accessing each B*-tree[1] to fetch the TID-lists and carry out the intersection, e.g., $L(f_1 = x) \cap L(f_2 = \emptyset) \cap ... \cap L(f_n = z)$. This approach would result in $C_R = (m+1) \cdot \left(n \cdot \left(h_F + \left(\frac{l_F \cdot e}{p}\right)\right)\right)$ page references. A closer inspection of the TID-lists used to perform the tests shows that at most $2 \cdot n$ TID-lists are involved in all (possibly $2^n - 1$) queries on C (a defined value and the null value for each f_i). Hence, keeping the TID-lists in a working buffer limits the cost to $C_R = 2 \cdot \left(n \cdot \left(h_F + \left(\frac{l_F \cdot e}{p}\right)\right)\right)$ page references in the worst case. Thus, the support of `MATCH PARTIAL` with the option `RESTRICT` for the "Delete from P" operation costs

1. The null value is considered as a special value, that is, null is used as a key value in all B*-trees of the F-key.

$$C_R = 2 \cdot n \cdot \left(h_F + \left(\frac{l_F \cdot e}{p}\right)\right) + n \cdot \left(h_K + \left(\frac{l_K \cdot e}{p}\right)\right) \quad .$$

Up to now we have not elaborated these check procedures regarding the sizes of B*-trees occurring in practical applications. A dependency analysis between the number of tuples N and the number of key attributes n shows a major problem of the sketched approach: Given a parent table with 10^6 tuples and 10 children per parent (resulting in about 10^7 child tuples) with a primary/foreign key made up out of three attributes with independent and uniform value distribution, we obtain 100 different values per attribute, i.e., the B*-trees only have 100 entries. This small number of entries has a significant impact on the partial match results of queries with only one attribute: In the parent relation, such a query results in 10^4 tuples and in the child relation up to 10^5 tuples, i.e., the TID-lists in the leaf pages of the corresponding B*-trees are very long[1]. If we assume that we manage these results as lists of TIDs with an entry length of 5 bytes these numbers result in $l_K \cdot e = 5 \cdot 10^4$ bytes (roughly 50 Kbytes) resp. $l_F \cdot e = 5 \cdot 10^5$ bytes (about 500 Kbytes). While the former is manageable within a multi-user environment, the latter is hardly possible. Hence, this approach of using n B*-trees seems only conceivable for the parent relation. This, however, is no severe problem because the thus supported partial match queries are needed for the parent only.

To remove this difficulty, we propose a hybrid approach: a compound B*-tree for the F-key of C and n single attribute B*-trees for the K-key of P. By this combination, the cost for inserting a tuple into C is reduced to $C_R = h_F + (n-u) \cdot \left(h_K + \left(\frac{l_K \cdot e}{p}\right)\right)$. The cost of “Delete from P” are given by

$C_R = (m+1) \cdot h_F + n \cdot \left(h_K + \left(\frac{l_K \cdot e}{p}\right)\right)$. Note that in this case $2 \cdot n$ is not an upper bound for m which, in turn, may range up to $2^n - 2$. Furthermore, we assume that the TID-lists in the B*-tree of the child relation do not exceed one leaf page.

In principle, the access path for the K- and F-key could be combined using the CAPS approach. However, such a combination is useless or even impractical: a CAPS for the compound keys does not provide any improvement concerning `MATCH PARTIAL`. Used for the n single B*-trees, it quickly suffers from unmanageable TID-lists.

Use of grid files

So far, we have simulated multi-key access and partial match queries to the relations P and C by “linear” access paths, that is, B*-trees designed for one-dimensional access. In order to investigate the question whether or not multi-dimensional access paths are better suited for checking the demands of the `MATCH`

1. In the sketched situation the B*-trees degenerate to inverted lists, because the height of those trees won’t exceed 1.

PARTIAL option, we consider the grid file [NHS84] as the best known multi-key access structure. The mapping principle of the grid file is sketched in Fig. 6 for two dimensions.

The point objects in data space D are mapped by means of the grid directory GD into the buckets of the grid file. For each of the n dimensions originating from the n attributes (keys), the grid file offers symmetric and uniform access thereby guaranteeing a balanced access structure independent from key distribution as well as insertion and deletion sequences.

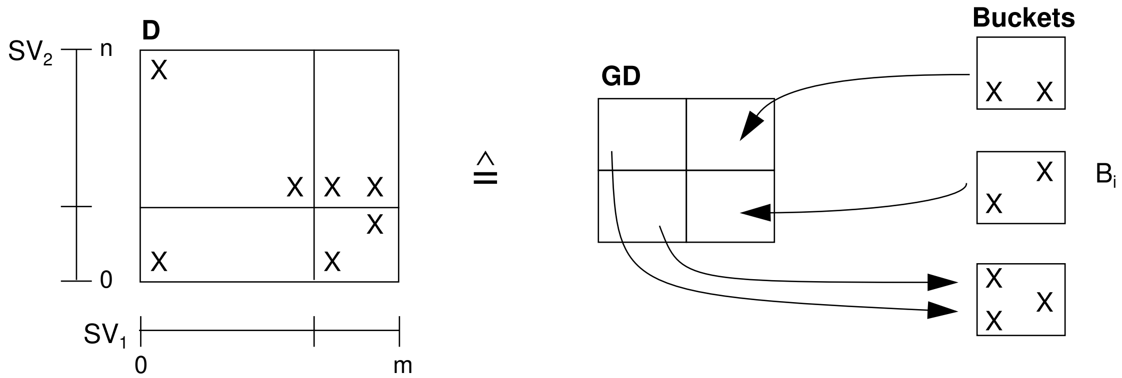


Fig. 6: Mapping principle of a grid file

The dimensions of D are represented by the ordered values of the attributes k_i and f_i resp. ($i \leq n$); null is considered as a special value for f_i attributes. Each dimension is described by a scale vector SV_i used to map the search predicate of a query to the GD. The set of specified scale values qualifies the GD entries which point to the buckets containing the records meeting the search condition. In order to compare the structure with the B*-tree, we assume that the buckets exclusively store the K- or F-key values (with n components) together with the corresponding TID or TID-list. Furthermore, we assume uniform distribution of all k_i and f_i attribute values ($i \leq n$).

In order to describe the cost of accessing a grid file some additional parameters are required:

S_i	number of scale values (intervals) in SV_i
E_{GD}	number of GD entries: $E_{GD} = \prod_{i=1}^{n} S_i$
B	number of buckets B_i: $B = N / (b * \beta_{avg})$[1] where b is the bucket size and β_{avg} the average load factor of a bucket
α	average number of GD entries mapping to a bucket, $\alpha \geq 1$: hence, $E_{GD} = \alpha * B$
P_{GD}	number of pages covered by GD: $P_{GD} = E_{GD} * e_{BID} / p$ where e_{BID} is the length of a bucket-ID and p the page size

The n scale vectors SV_i are represented as one-dimensional arrays; they are always kept in main memory such that they do not provoke extra page references. Our access model further assumes that a point

1. A ceiling function has to be applied to the computed access cost or storage size because disk access or pages (buckets) cannot occur in fractions.

query requires a single disk access to the GD. For partial match queries, however, sets of GD entries, which may be mapped to pages in sophisticated ways, have to be located. To reflect this mapping in our access model, we use the following heuristic approximation for the cost of GD access: $C_{GD} = P_{GD} * u/n$. The set of qualified GD entries determines the number of buckets to be selected (by applying the given α). Moreover, all buckets fully contain the allocated key/TID or key/TID-list pairs.

Apparently, the lion's share of the query processing costs using grid files is caused by the set of buckets to be accessed. For GD_P, there are only two kinds of search-key terms: $k_i = v$ and k_i ='–'; for a don't-care value in the search predicate, all existing values of the resp. key qualify. Hence, a point query delivers a single GD entry. Partial match queries with one don't-care value (k_i ='–') select S_{Pi} GD entries; two don't-care values k_i and k_j lead to $S_{Pi} \cdot S_{Pj}$ GD entries and so on.

"Insert into C" with u undefined values in the F-key requires a partial match query on GD_P with u don't-care values in the attributes of key K. If the set of attributes k_i is indexed by m_i, i = 1,..., u, we obtain the following costs of page references (worst case)

$$C_R = P_{GD} \cdot \frac{u}{n} + \frac{\prod_{i=1}^{u} S_{m_i}}{\alpha} \quad ,$$

where the first term stands for the GD access and the second one for fetching the buckets. If the number of scale values S is equal in all dimensions, the cost formula can be simplified to the following form

$$C_R = P_{GD} \cdot \frac{u}{n} + \frac{S^u}{\alpha} \quad ,$$

which makes clear the dominant influence of parameter u. For our convenience, we will use this simplified formula in the following; however, it may deliver only approximate numbers of page references.

To make the cost factors clear, assume the following situation: $N=10^6$, b=200, $\beta_{avg}=0.75$, S=20, u=2, and $\alpha = 1.2$, then we potentially reference $C_R = (7 + 334)$ pages to check whether there is a partially matching tuple t_P for the inserted tuple t_C. Since we can stop the evaluation of the buckets as soon as we have found a valid tuple t_P, the given cost formula describes the worst case. To complete the "Insert into C" operation, we have to locate the insertion point for t_C in GD_C which needs a point query (2 page references).

"Delete from P" is the second critical operation which has to be supported by partial match access.The inverse check procedure is applied to test the existence of tuples having partially matching foreign keys. If we execute a partial match query, e.g. <x, –, –>, on GD_P, all keys can be derived including the defined component x. For this purpose, all buckets qualified by <x, –, –> have to be accessed and filtered (using the x-value in our example). The list of keys derived allows for all templates with component x to test whether or not the parent exists. Optimization requires to cache the list of keys in main memory; otherwise the full set of partial match queries has to be applied sequentially. Hence, by applying our optimized search and check procedure we obtain the following cost formula for page references:

$$C_R = n \cdot \left(P_{GD} \cdot \frac{n-1}{n} + \frac{\prod_{i=1}^{n-1} S_{m_i}}{\alpha} \right) \quad \text{or} \quad C_R = n \cdot \left(P_{GD} \cdot \frac{n-1}{n} + \frac{S^{n-1}}{\alpha} \right) \text{ for uniform numbers of all } S_i.$$

Unsuccessful inspections in GD_P require checks in GD_C to determine whether children exist for the resp. templates. Since we assume m such tests, m + 1 point queries have to be performed using GD_C resulting in $C_R = (m + 1) \cdot 2$ page references.

Comparison of access path structures

Table 5 compares the search costs for the basic support of `MATCH PARTIAL` with the option `RESTRICT`. Remember, these costs represent the required accesses to determine which children are subject to referential actions, but not the entire costs to accomplish them (such an analysis would have to take the referential action `SET DEFAULT` into consideration to deliver worst case costs).

	2 compound-key B*-trees	n simple key B*-trees for P compound B*-tree for C	2 grid files
Insert into P	h_K	$n \cdot h_K$	2
Delete from P	$h_K + (h_K + N_{Kleaf} - 1) + (m+1) \cdot h_F$	$n \cdot \left(h_K + \left(\frac{l_K \cdot e}{p}\right)\right) + (m+1) \cdot h_F$	$2 + n \cdot \left(P_{GD} \cdot \frac{n-1}{n} + \frac{S^{n-1}}{\alpha}\right) + 2 \cdot (m+1)$
Update attribute k_i of P	$h_K + (h_K + N_{Kleaf} - 1) + (m+1) \cdot h_F + h_K$	$n \cdot \left(h_K + \left(\frac{l_K \cdot e}{p}\right)\right) + (m+1) \cdot h_F + n \cdot \left(h_K + \left(\frac{l_K \cdot e}{p}\right)\right)$	$2 + n \cdot \left(P_{GD} \cdot \frac{n-1}{n} + \frac{S^{n-1}}{\alpha}\right) + 2 \cdot (m+1) + 2$
Insert into C	$h_K + (h_K + N_{Kleaf} - 1)$	$h_F + n \cdot \left(h_K + \left(\frac{l_K \cdot e}{p}\right)\right)$	$2 + \left(P_{GD} \cdot \frac{u}{n} + \frac{S^u}{\alpha}\right)$
Delete from C	h_F	h_F	2

Table 5: Summary of the results (page references) for `MATCH PARTIAL` with the option `RESTRICT`

	2 compound-key B*-trees	n simple key B*-trees for P compound B*-tree for C	2 grid files
Update attribute f_i of C	h_F + $(h_K + N_{Kleaf} - 1)$ + h_F	h_F + $n \cdot \left(h_K + \left(\frac{l_K \cdot e}{p}\right)\right)$ + h_F	2 + $\left(P_{GD} \cdot \frac{u}{n} + \frac{S^u}{\alpha}\right)$ + 2

Table 5: Summary of the results (page references) for `MATCH PARTIAL` with the option `RESTRICT`

Obviously, a comparison of the search costs is difficult at the chosen level of abstraction due to the fact that some cost factors apply to only one or two of the implementation alternatives (e.g. N_{Kleaf} applies only to the compound-key solution). To get some hints about the relative costs of the `MATCH PARTIAL` support, we elaborate some practical cases by using numbers approximating large applications (in the order of magnitude of Sect. 4.1):

- The number of tuples N_P is 10^6, and the K-keys resp. F-keys consist of n=3 attributes. We use u=2 for the "Insert into C" operation.
- A key/TID-pair needs 10 bytes if the key is simple or 15 bytes if it is compound. In both cases, we assume key compression. Note that the front/rear-compression is not applicable for grid files. Therefore, the key/TID-pairs managed in the buckets of a grid file are assumed to have a length of 25 bytes. The page or bucket size is 4K bytes. Further parameters: β_{avg} = 1, e_{BID} = 4 bytes.
- $E_{GD} = S^3 = \alpha$*B; a minimum value of S is chosen which also minimizes α for the given B.
- The heights of the B*-trees are derived from Fig. 4. For the computation of N_{Kleaf}, completely filled leaf pages are assumed (best case!).

The grid-file performance depends heavily on the number of buckets and directory entries. Here, we suppose the best mapping of GD to the buckets minimizing the number of buckets and GD entries. Nevertheless, we obtain substantial costs as shown in Table 6. Note, β_{avg} = 0.75 would increase the number of page references roughly by 21%.

	2 compound-key B*-trees	n simple key B*-trees for P compound B*-tree for C	2 grid files
Delete from P	$3759 + 4 \cdot m$ 3759 -- 3783	$46 + 4 \cdot m$ 46 -- 70	$1006 + 2 \cdot m$ 1006 -- 1018
Insert into C	3755	46	336

Table 6: Exemplary access costs for referential integrity with `MATCH PARTIAL`

As discussed previously, the inverse and the straight-forward check procedures provoke the same cost in the worst case ($m = 2^n - 2$). Table 5 and 6 reveal, however, that the lion's share of the page references arises from the partial match queries. Therefore, it seems to be advisable in practical cases to execute the cheap point queries first thereby hoping to find no or only a few foreign keys. This may greatly reduce the number of partial match queries required.

The solution based on two compound B*-trees is not competitive at all, because the cost of the leaf page scan grows linearly with N. Only for very small N or for special partial match search keys (having the leftmost values defined) this solution would be a good contender for the task considered.

Our best solution relies on n single B*-trees for the partial match queries and on a compound B*-tree for the point queries. A combination based on a CAPS solution (the best alternative for the regular MATCH clause) is here unfeasible because of extreme TID-list lengths for larger numbers of N. As indicated in Table 6, for the given scenario the other two approaches are outperformed by factors of 15-20 resp. 50-80. Nevertheless, our hybrid approach remains expensive, that is, the usage of `MATCH PARTIAL` seems prohibitive in any time-critical application (e.g. OLTP). Note that in our cost measures, we have neglected the computation costs for the TID-list intersections. These costs, however, will become substantial if the lists grow beyond some threshold which, in turn, depends on other parameters (e.g. hardware capabilities), and, therefore, this alternative may reach its limits, too.

The most elegant approach is the usage of an access path which supports the costly partial match queries directly. As an example we presented the grid file. In contrast to the expected result, however, the analysis obtained relatively bad numbers for the grid file performance. This is mainly caused by the fact that we are not interested in all resulting tuples of a given partial match query, but only on the information whether or not at least one tuple exists. While the former is the classical application for grid files the latter is not. In addition to the performance argument, other problems are yet to be solved to provide grid files for large applications: Referential integrity maintenance is typically performed in multi-user environments with a high degree of concurrent access. To cope with such situations, optimal locking protocols were designed for B*-trees [ML92, Mo90], giving direct access for keys and key ranges whereas competitive locking protocols for grid files [Sa86] are not known so far. For this reason and the performance figures derived, our best candidate to support `MATCH PARTIAL` remains the hybrid solution based on B*-trees. Nevertheless, our best advise is to avoid the use of `MATCH PARTIAL` at all.

5 Conclusions and Outlook

We have presented an investigation of referential integrity support in relational DBMS. The focus of our paper has primarily been on determining the functional requirements of referential integrity maintenance caused by modification operations on the parent relation P and the child relation C. Furthermore, an extensive study has been performed to answer the question: “which access paths should be provided in a DBMS to effectively and efficiently meet these functional requirements?”

Our initial discussion outlined the specification of the SQL2 standard and its semantics as far as referential integrity is concerned. As an outcome, we have derived the query types which are necessary to maintain referential integrity. If the regular MATCH option is used, then the complexity of all queries required is at most of type (**P**, **S**) which represents a point query in the key space and results in a set of elements (TIDs or tuples). This type of query is well supported through a B*-tree (either for the foreign key or for the primary key). An optimization can be achieved using only a single CAPS jointly used for the primary key and the foreign key.

This relatively simple situation gets much more complicated if the `MATCH PARTIAL` option of the referential integrity constraint definition is considered. In such cases, the query type needed is ($\mathbf{PM_u}$, **E**) which denotes a partial match query (with u unknown values) in the resp. key space resulting in a set of tuples or TIDs. Another complication arises through the exponential growth of the number of point queries to be tested. As it turns out, the latter does not contribute the major share to the costs of all access paths explored. Therefore, support of partial match queries becomes the most critical factor. For this reason, the solution based on compound keys is inappropriate. Although the access costs using a grid file are very low for some operations, others are remarkably more expensive than those of the hybrid solution based on B*-trees. Accordingly, we recommend the latter solution when `MATCH PARTIAL` is used.

The 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. To do so existing applications have to be evaluated to reveal the practical relevance of `MATCH PARTIAL`. Furthermore, it may be interesting to analyze real world applications to see whether or not the various MATCH options interfere on one parent relation. For such cases, the combined usage of our concepts has to be investigated.

Acknowledgements

We would like to thank C. Huff, E. Rahm, and H. Schöning for their helpful comments on an earlier version of this paper. The comments and questions of the referees are appreciated.

6 References

BCE77 Blasgen, M.W., Casey, R.G., Eswaran, K.P.: An encoding method for multified sorting and indexing. In: Comm. ACM. Vol. 20. No.11. Nov. 1977. 874-876

Co70 Codd, E.F.: A Relational Model of Data for Large Shared Data Banks, in: Comm. ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

Co79 Comer, D. The Ubiquitous B-Tree, in: ACM Computing Surveys, Vol. 12, No. 2, 1979, pp. 121-137.

Da81 Date, C.J.: Referential integrity, in: Proc. 7th Int. Conf. on VLDB, 1981, pp. 2-12.

Da90 Date, C.J.: Relational Databases: Selected Writings 1985-1990, Addison-Wesley Publ. Comp., 1990.

Gr78 Gray, J.N.: Notes on Data Base Operating Systems, in: Lecture Notes Computer Science, 60, Operating systems: An advanced course, Springer-Verlag, 1978, pp. 393-481.

Hä78 Härder, T.: Implementing a Generalized Access Path Structure for a Relational Data Base System, in: ACM TODS, Vol. 3, No. 3, 1978, pp. 285-298.

Ma91 Markowitz, V.M.: Safe Referential Integrity Structures, in: Proc. 17th Int. Conf. on VLDB, 1991.

ML92 Mohan, C., Levine, F.: ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging, in: Proc. ACM SIGMOD, San Diego, 1992, pp. 371-380.

Mo90 Mohan, C.: ARIES/KVL: A Key-Values Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes, in: Proc. 16th Int. Conf. on VLDB, Brisbane, Australia, August 1990.

NHS84 Nievergelt, J., Hinterberger, H., Sevcik, K.C.: The Grid File: An Adaptable, Symmetric Multi-key File Structure, in: ACM TODS, Vol. 9, No. 1, 1984, pp. 38-71.

NMR79 Nevalainen, O., Muurinen, K., Rantala, S.: A note on character compression. Angewandte Informatik. 21:7. Vieweg-Verlag, 1979. 313-318

Re93 Reinert, J.: Ensuring Referential Integrity in SQL2 and SQL3, Internal report, University of Kaiserslautern, Department of Computer Science, 1993.

Sa86 Salzberg, B.: Grid File Concurrency, in: Information Systems, Vol. 11, No. 3, 1986, pp. 235-244.

Sh90 Shaw, P.: Database Language Standards: Past, Present, Future, in: Database Systems of the 90s, A. Blaser (ed.), LNCS 466, Springer-Verlag, 1990, pp. 50-88.

SQL92 ISO/IEC 9075:1992, Database Language - SQL, July 1992.

SQL3 X3H2-93-091/ YOK-003 ISO/IEC JTC1: ISO/ANSI Working Draft - Database Language SQL3, 02.1993.

Va87 Valduriez, P.: Join Indices, in: ACM TODS, Vol. 12, No. 2, 1987, pp. 218-246.

Wa73 Wagner, R.E.: Indexing design considerations. IBM Sys. J. 12:4. 1973. 351-367