

Constraints

Constraints are fundamental to databases and application programming. Unfortunately, in the programming industry the discussion about constraints often degrades to a rather shallow dilemma:

Should the constraints be enforced in the database, or in the application / middle tier?

Well, if the reader is still undecided about this¹, then it makes little sense to continue. Constraint implementation in the database matured to a pretty sophisticated level, not the least of which should be credited to the wealth of the underlying language – SQL.

As the reader is assumed to be familiar with the basics of database constraints – unique key, referential integrity, *check* constraint – in this chapter we'll venture into an obscure area of complex constraints. SQL standard allows declaring complex constraints as *ASSERTIONS*, but no database vendor supports them. For quite some time database triggers were the only complex constraint enforcement technique. We'll help the reader to broaden the view. Certain constraints can be implemented with the *Function Based* method. It might be not as general as the materialized views based technique, but is certainly not short of elegance. The materialized views based method, on the other hand, is somewhat elaborate, although much more powerful. It is our favorite in this chapter.

Function Based Constraints

¹ As Marshall Spight put it: "I will not forgive you ☺ if you don't at least mention a few of the worst problems with application-enforced constraints:

- 1) not declarative
- 2) not centrally enforced—may be bypassed
- 3) application-language specific!"



Function based constraints require little introduction. Sooner or later every database person comes across a unique constraint on `UPPER(person.name)`. The other popular example of function based constraint is

```
alter table Emp
ADD CONSTRAINT namesInUppercase CHECK ( UPPER(ename)=ename )
```

In this chapter we'll give many reasons why declarative constraints declaration is always better than any alternative solution. I witnessed that in my own experience. Consider a query

```
select * from emp
where ename like 'MIL%'
```

When I saw this query execution plan

OPERATION	OBJECT NAME
⊞ SELECT STATEMENT	
⊞ TABLE ACCESS	EMP
⊞  Filter Predicates	
⊞  AND	
⊞ ENAME LIKE 'MIL%'	
⊞ UPPER(ENAME) LIKE 'MIL%'	

I was puzzled where did the second conjunct `UPPER(ename) like UPPER('MIL%')` come from? The plan looked as if the optimizer rewrote the query into

```
select * from emp
where ename like 'MIL%'
and UPPER(ename) like 'MIL%'
```

The answer maybe immediate here in the hindsight of the constraint that I declared earlier, but in practice I was totally forgotten about the constraint, which I declared a while ago. After quick investigation, I indeed found that there was a check constraint `UPPER(ename) = ename` declared upon the `Emp` table. In other words, RDBMS engine leverages constraints when manipulating query predicates.

What is the point of adding one more filter condition to a query that would execute correctly anyway? Imagine that we add a function-based index upon `UPPER(ename)` pseudo column. Then, it might be used, even though the original query

```
select * from emp
where ename like 'MIL%'
```

doesn't refer to any function within the predicate. It doesn't need to be mentioned that query execution leveraging an index often means difference between life and death from performance perspective.

In the next section we'll experiment with somewhat more elaborated function based constraint.

Symmetric Functions

Consider an inventory database of boxes

```
table Boxes (  
    length integer,  
    width  integer,  
    height integer  
)
```

Box dimensions in the real world, however, are generally not given in any specific order. The choice what dimensions becomes length, width, and height is essentially arbitrary. What if we want to **identify** the boxes according to their dimensions? For example, we would like to be able to tell that the box with `length=1`, `width=2`, and `height=3` is the same box as the one with `length=3`, `width=1`, and `height=2`. Furthermore, how about declaring a *unique dimensional* constraint? More specifically, we won't allow any two boxes that have the same dimensions.

An analytical mind would have no trouble recognizing that the heart of the problem is the column ordering. The values of the `length`, `width`, and `height` columns can be interchanged to form another legitimate record! Therefore, why don't we introduce 3 pseudo columns, say `A`, `B`, and `C` such that

$$A \leq B \leq C$$

Then, a unique constraint on `A`, `B`, `C` should satisfy our requirement! It could be implemented as a function based unique index, as long as we can express `A`, `B`, `C` analytically in terms of `length`, `width`, `height`. Piece of cake: `C` is the *greatest* of `length`, `width`, `height`; `A` is the *least* of them, but how do we express `B`? Well, the answer is easy to write

```
B = least (greatest (length,width),  
           greatest (width,height),  
           greatest (height,length) )
```

although difficult to explain.

A mathematical perspective, as usual, clarifies a lot. Consider cubic equation

$$x^3 + a x^2 + b x + c = 0$$

If we know the roots x_1, x_2, x_3 then, the cubic polynomial could be factored, so that we have

$$(x - x_1)(x - x_2)(x - x_3) = 0$$

Marrying both equations we express coefficients a, b, c in terms of roots x_1, x_2, x_3

$$a = -x_1 - x_2 - x_3$$

$$b = x_1 x_2 + x_2 x_3 + x_3 x_1$$

$$c = -x_1 x_2 x_3$$

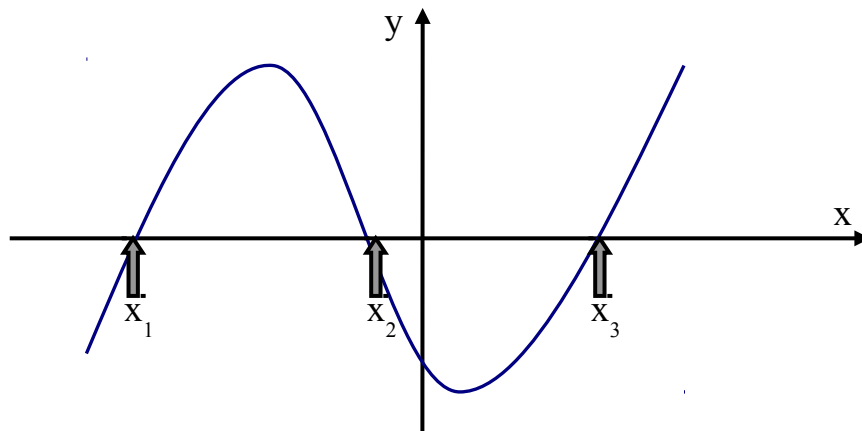


Figure 4.1: A shape of the graph of the polynomial $y=(x-x_1)(x-x_2)(x-x_3)$ is entirely defined by the roots x_1, x_2 , and x_3 . Exchanging them doesn't affect anything.

The functions $-x_1-x_2-x_3$, $x_1x_2+x_2x_3+x_3x_1$, $-x_1x_2x_3$ are **symmetric**. Permuting x_1, x_2, x_3 has no effect on the values a, b, c . In other words, the order among the roots of cubic equation is irrelevant: formally, we speak of a set of roots, not a list of roots². This is exactly the effect we want in our example with Boxes. Symmetric functions rewritten in terms of length, width, height are

² Moreover, the roots are generally complex, so that the order is meaningless.

```
length+width+height
length*width+width*height+height*length
length*width*height
```

Those expressions were simplified a little by leveraging the fact that the negation of a symmetric function is also symmetric.

Our last solution is strikingly similar to the earlier one, where the *greatest* operator plays the role of multiplication, while the *least* operator goes as addition. It is even possible to suggest a solution, which is a mix-in between the two

```
least(length,width,height)
least(length+width,width+height,height+length)
length+width+height
```

A reader can check that these three functions are again symmetric³.

The last step is recording our solution in formal SQL

```
table Boxes (
  length integer,
  width integer,
  height integer
);

create unique index b_idx on Boxes(
  length + width + height,
  length * width + width * height + height * length,
  length * width * height
);
```

Symmetric functions provide a basis for a nifty solution. In practice however, a problem can often be solved by schema redesign. In the box inventory database example, we don't even need schema redesign: we can just require to change the practice of inserting unconstrained records *(length,width,height)*, and demand that

```
length ≥ width ≥ height
```

Materialized View Constraints

Base relations – tables – and derived ones – views – are fundamental building blocks of Relational databases. A derived relation may be *virtual*, meaning that the defining relational

³ Mathematically inclined readers are referred to the field of *Tropical* arithmetic, where the *least* operator plays the role of ordinary addition, and addition is taken the role of ordinary multiplication. This is how we've got the last set of symmetric functions – by just rewriting symmetric polynomials in tropical arithmetic.

expression is evaluated in terms of the base relations, or *materialized*, meaning that the relation is actually stored. In database practice they are commonly referred to as (plain) views and materialized views, correspondingly.

In many respects a materialized view is similar to a (base) table. One can index it, declare a constraint, even (heaven forbid) associate a trigger. Declaring constraints upon materialized views turns out to be a very powerful method of enforcing complex constraints with limited SQL support.

In the database research literature leveraging materialized views for constraint enforcement has been suggested as early as in 1978-1979⁴. Given that it was a long time before materialized views became a reality in practical RDBMS implementations, those ideas had to remain dormant. It was Tony Andrews who sparked a renewed interest in materialized view constraint enforcement in the Oracle community.

When declaring a constraint a starting point is expressing it as a formal expression. Consider the foreign key integrity constraint, for example

For any record in the Emp table there has to be a matching Dept record

This constraint declaration is informal because of the term *matching*. Elaborating it we get more precise statement

For any record *e* in the Emp table there has to exist a Dept *d* such that *e.deptno* = *d.deptno*

It is generally a good idea to rephrase a constraint as *impossible* condition. This doesn't really change anything from a logical perspective, but in real life law enforcement implies some real world action, which has to be invoked whenever the law is broken. In our example

There doesn't exist a record *e* in the Emp table such that there is no Dept *d* such that *e.deptno* = *d.deptno*

⁴ Michael Hammer, Sunil Sarin. *Efficient Monitoring of Database Assertions*. ACM SIGMOD International Conference on Management of Data, 1978.

Peter Buneman, Eric Clemons. *Efficiently Monitoring Relational Databases*. ACM Transactions on Database Systems, Vol. 4, No. 3, Sept 1979, pages 368-382

Admittedly, this sentence sounds awful (even for me, who wrote it in the first place). I can imagine how a reader without background in logic might feel. What is the point of elaborating formal content, if at the end we produce such gibberish as the last statement?

Well, this situation is common in math. We transform an expression in a series of steps, so that expression might become ugly in the process, but sometimes we get lucky and it finally collapses into a simple formula. In our example, it takes one more step

Consider all the record *e* in the *Emp* table such that there is no *Dept d* with *e.deptno* = *d.deptno*. There mustn't to be any!

before the statement finally contracts to

The difference between the set of the *Emp.dept* and *Dept.dept* is empty.

It is formal, because we can express it in SQL, where the only missing part is equality between views

```
select deptno from Emp
minus
select deptno from Dept
=  $\emptyset$ 5
```

Let's refer to the view on the left side as *EmpWithoutDept*.

On afterthought, this assertion is obvious. The *Emp-Dept* referential integrity constraint is violated only if the *EmpWithoutDept* view becomes nonempty. It could happen when a tuple is inserted into the *Emp* table, or deleted from the *Dept* table.

⁵ If the reader is uncomfortable with the empty set symbol \emptyset , you can think of it as shorthand for

```
select deptno from dept where 1=0
```

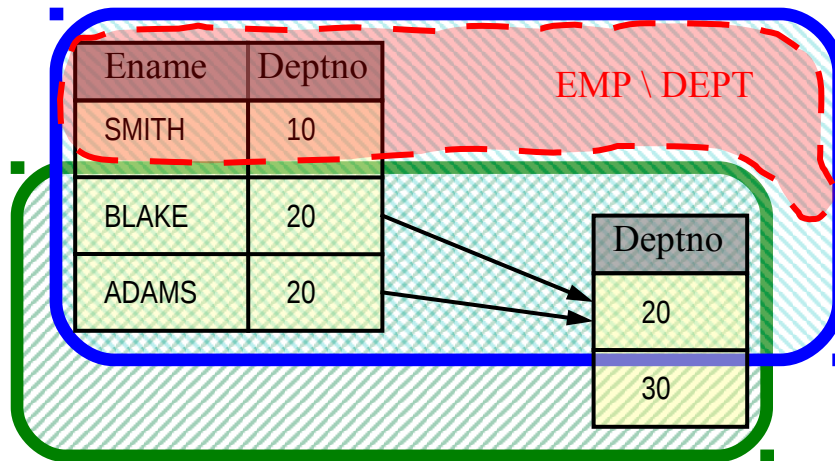


Figure 4.2: The record (Ename=SMITH, Deptno=10) doesn't match to any record in the department table: it belongs to the set EMP \ DEPT. The foreign key constraint asserts EMP \ DEPT = \emptyset .

Since there is no concept of equality between views in SQL, we have to express the condition for emptiness by other means. If the `EmpWithoutDept` view is not empty, then `EmpWithoutDept.dept` is not null! A simple check constraint would do:

```
CREATE MATERIALIZED VIEW EmpWithoutDept AS
select deptno from Emp
minus
select deptno from Dept;

ALTER TABLE emp_minus_dept_mv
ADD CONSTRAINT EWD_is_empty CHECK( deptno is null );
```

Whenever referential integrity constraint is violated, `EmpWithoutDept` view becomes nonempty, which triggers violation of the `EWD_is_empty` constraint.

While (plain) views don't require any maintenance, materialized views need to be kept up to date with the base relations. Efficient update of materialized view is achieved via *incremental evaluation*⁶.

⁶ Again, terminology used by database vendors varies. Oracle, for example, calls incrementally updateable materialized views *fast refreshable*.

Incremental Evaluation

Queries and updates often don't get along with each other. They are conflicting goals from a performance perspective. We can speed up some queries at the cost of introducing some auxiliary structures. The most familiar examples of such structures are indexes and materialized views, and there are other cases which warrant a general concept.

Those structures need to be kept up to date with the base tables. Completely reevaluating them is out of question for any sizeable database. They have to be maintained incrementally: a change to the structure is small when the update transaction is small, which makes the overall performance acceptable. Incremental query evaluation is one of the most important performance ideas in the database world.

In practice we have to deal with database implementations which dictate seemingly arbitrary limitations on what operations do support incremental refresh and which don't. Oracle circa 2005 doesn't support set operators in the definition of incrementally maintained materialized view, for example. We could work around this limitation by rewriting the assertion in an equivalent form that leverages only supported operators.

```
CREATE MATERIALIZED VIEW EmpOuterJoinDept
REFRESH FAST ON COMMIT AS
select d.deptno ddept, d.rowid drid, e.rowid erid
from Emp e, Dept d
where e.deptno=d.deptno(+);

ALTER TABLE EmpOuterJoinDept
ADD CONSTRAINT ck_oj_mv CHECK(ddept is not null);
```

However, one especially attractive feature is lost: this materialized view is no longer empty.

Trigger solution is unreliable

A comparison between the materialized view and trigger based solution doesn't favor the latter. First, the trigger has to cover all operations: insert, update and delete – missing any of them would allow integrity violation. Second, a constraint involving more than one table has to be covered by multiple triggers (in our example, on both `Emp` and `Dept` tables).

More important, however, is that writing triggers is a challenging exercise from concurrency semantics perspective. It is better to delegate complex code to RDBMS engine developers, and leverage the high level features that RDBMS offers.

In general, constructing incrementally updateable materialized view with such limitations becomes a coding exercise that tests your patience. It is almost always possible to express an operator in a chain of incrementally refreshable materialized views. Here is how `minus` operator can be represented, for example

```
create materialized view empDepts as
select deptno, count(*) cnt
from emp
group by deptno;

create materialized view deptEmpCross as
select dept.deptno dd, empDepts.deptno ed, dept.rowid drid,
empDepts.rowid erid
from empDepts, dept;

create materialized view deptEmpJoin as
select dept.deptno dd, empDepts.deptno ed, dept.rowid drid,
empDepts.rowid erid
from empDepts, dept
where dept.deptno=empDepts.deptno;

create materialized view deptEmpUnion as
```

```
select '1' marker,dd,rowid rid
from deptEmpCross
union all
select '2' marker,dd,rowid rid
from deptEmpJoin;

create materialized view deptCounts as
select dd, count(*) c
from deptEmpUnion
group by dd;

create materialized view empDeptCount as
select count(*) c
from empDepts;

create materialized view Final as
select dd, c1.rowid rid1, c2.rowid rid2
from empDeptCount c1, deptCounts c2
where c1.c=c2.c;
```

The basic idea is that set difference can be expressed via joins and aggregation with counting, and incremental refresh of the latter operations is supported. This is mostly pointless from a practical perspective, however. It is hard to imagine that anybody would buy the overhead of 7 (!) nonempty materialized views for a mere set difference implementation.

Sooner or later incremental refresh limitations will be lifted. For the purpose of further constraint study in this book, let's continue pretending as if it already happened.

Disjoint Sets

Enforcing that the two tables have no common records is a very common practical problem. Consider a typical Object-Oriented design scenario where the `Employee` type has two subtypes: `FullTimeEmp` and `PartTimeEmp`. On the database side, suppose that there are only two tables: `FullTimeEmp` and `PartTimeEmp`. How do we enforce uniqueness of an employee `id`? Certainly, we can declare both `FullTimeEmp.id` and `PartTimeEmp.id` to be unique, but how do we guarantee that there is no employee, who is both full time and part time? These sets have to be disjoint

```
CREATE MATERIALIZED VIEW fullTime_intersect_partTime
select id from FullTimeEmp
intersect
select id from PartTimeEmp

ALTER TABLE fullTime_intersect_partTime
ADD CONSTRAINT disjointClasses CHECK(id is null)
```

There is another, more straightforward, but somewhat cumbersome way to approach this problem. Like any other constraint, a unique key can be enforced via a materialized view as well. Let's start with the smaller and easier task of defining `FullTimeEmp.id` unique key. Suppose that `FullTimeEmp` has two more columns: `name` and `startedAt`. Again, we begin rephrasing the constraint as an impossible condition

The two employee records e_1 and e_2 are contradictory whenever $e_1.id = e_2.id$, and yet either $e_1.name \neq e_2.name$ or $e_1.startedAt \neq e_2.startedAt$. No contradictory employee records are allowed.

It easy to express this constraint formally

```
CREATE MATERIALIZED VIEW contradictoryEmployees
select e1.id id from FullTimeEmp e1, FullTimeEmp e2
where e1.id = e2.id and (
    e1.name <> e2.name or e1.startedAt <> e2.startedAt
)

ALTER TABLE fullTime_intersect_partTime
ADD CONSTRAINT disjointClasses CHECK(id is null)
```

It is obvious, that the number of inequality comparisons would grow with the number of columns in the table. This is why I referred to this approach as cumbersome. Some RDBMS implementations introduce a `ROWID` pseudo column, which is guaranteed to be unique. This allows comparing `ROWIDS` instead of actual values:

```
CREATE MATERIALIZED VIEW contradictoryEmployees
select e1.id id from FullTimeEmp e1, FullTimeEmp e2
where e1.id = e2.id and e1.rowid <> e2.rowid

ALTER TABLE fullTime_intersect_partTime
ADD CONSTRAINT disjointClasses CHECK(id is null)
```

Without `ROWIDS` our constraint is actually weaker than a unique key, as it can't distinguish duplicate records. Duplicate records don't exist in set semantics, but SQL operates with bags. One extra nested materialized view is needed to exclude duplicates.

We are one step from the final solution. Just declare the unique key constraint on a materialized view, which is the union of `FullTimeEmp` and `PartTimeEmp` tables. However, the `union` is not defined on tables which are *incompatible* and, in fact, it's easy to imagine that tables corresponding to different subclasses have different attributes. The concern about union being not defined for

incompatible relations has been already addressed in the previous chapter, where we introduced the outer union.

There is another reason why this solution is less elegant than the one with disjoint set constraint. The outer join view is not empty, which implies storage overhead. Although in principle the RDBMS engine could flatten nested materialized views, don't expect this to be implemented anytime soon.

Disjoint Intervals

Consider a list of intervals

```
table Intervals (
  head integer,
  tail integer
)
```

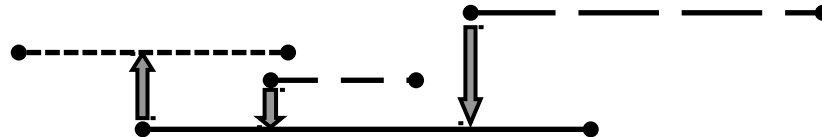
We would like to declare these intervals as mutually disjoint.

In math, disjoint intervals are defined as sets that don't intersect with each other⁷. This definition, however, isn't very useful in our situation, since we have to formulate the intervals disjoint condition in terms of intervals boundaries.

Once more, we begin rephrasing the constraint as an impossible condition

The intervals $[head_1, tail_1]$ and $[head_2, tail_2]$ overlap whenever $head_2$ is between $head_1$ and $tail_1$. No overlapping intervals are allowed.

Wait a minute, something must be wrong here! Interval overlapping condition has to be symmetric with respect to both intervals, while the formal statement that we wrote doesn't look symmetric at all. Indeed, what if interval $[head_2, tail_2]$ covers $[head_1, tail_1]$? Then $head_2$ is not between $head_1$ and $tail_1$, and yet intervals do overlap. This non-symmetry is illusory, however.



⁷ More precisely, intervals $[head_1, tail_1]$ and $[head_2, tail_2]$ intersect whenever there is a point x such that $x \in [head_1, tail_1]$ and $x \in [head_2, tail_2]$.

Figure 4.3: Overlapping intervals always have the head of one interval bounded between the ends of the other interval.

Let's write the constraint formally:

```
CREATE MATERIALIZED VIEW overlapping_intervals
select i1.head h
from intervals i1, intervals i2
where i2.head between i1.head and i1.tail

ALTER TABLE overlapping_intervals
ADD CONSTRAINT no_overlapping_intervals CHECK(h is null)
```

We see that *i1* and *i2* iterate over the set of all intervals, each pair of intervals *[a,b]* and *[c,d]* in the set would be considered twice: first time when *i1.head = a*, *i1.tail = b*, *i2.head = c*, *i2.tail = d*, and second time when *i1.head = c*, *i1.tail = d*, *i2.head = a*, *i2.tail = b*. This reasoning, however, exposes a bug in our implementation. Could *i1* and *i2* be the same interval? We have to add one more predicate that excludes this possibility:

```
CREATE MATERIALIZED VIEW overlapping_intervals
select i1.head h
from intervals i1, intervals i2
where i2.head between i1.head and i1.tail
and (i2.head <> i1.head or i2.tail <> i1.tail)

ALTER TABLE overlapping_intervals
ADD CONSTRAINT no_overlapping_intervals CHECK(h is null)
```

Alternatively, we could have used asymmetric join condition, and consider each pair of intervals once only. We define a *total* order among all the intervals so that for each pair of intervals *i1* and *i2* either *i1 precedes i2*, or *i2 precedes i1*. Lexicographical order comparison predicate

i2.head < i1.head or (i2.head = i1.head and i2.tail < i1.tail)

defines a total order. Then we can rewrite the constraint as follows:

Consider all the pairs of intervals such that *[head₁,tail₁]* precedes *[head₂,tail₂]*. They overlap whenever head₂ is less than or equal to tail₁. Again, no overlapping intervals are allowed.

Formally,

```
CREATE MATERIALIZED VIEW overlapping_intervals
select i1.head h
from intervals i1, intervals i2
where (i2.head < i1.head or
       i2.head = i1.head and i2.tail < i1.tail)
and i2.head <= i1.tail

ALTER TABLE overlapping_intervals
ADD CONSTRAINT no_overlapping_intervals CHECK(h is null)
```

Temporal Foreign Key Constraint

Audit trail is a database design where records are never deleted. All the data modifications are logged in temporal tables. Every record in a temporal table obtains two timestamp attributes: `CREATED` and `DELETED`. The values of the other attributes are valid during the interval starting with `CREATED` date and ending with `DELETED` date. Now that the same record of values is scattered into many records, how do we enforce constraints? Specifically, given two tables with parent-child relationship, how do we enforce referential constraint between their “temporalized” versions?

```
table HistParent (  
    id integer,  
    ...,  
    created date,  
    deleted date  
);  
  
table HistChild (  
    pid integer, -- foreign key to HistParent.id???  
    ...,  
    created date,  
    deleted date  
);
```

Let’s formulate the constraint, first informally in English, then in SQL. A child record can be created only if its parent record already exists. Likewise, a parent record can’t be deleted until it has at least one child. Informally,

A child lifespan must be contained within the parent lifetime

The critical issue is defining the parent and child lifetimes.

Since each parent is identified by the `id` attribute, it’s quite easy to define its lifetime. The lifetime of a parent is the longest span of time covered by the chain of `[created, deleted]` intervals. Now we can invoke the interval coalesce technique from chapter 1, and obtain the parent lifetime view

```
view ParentLifetime (  
    id integer,  
    birth date,  
    death date  
);
```

Please note that all the attributes marked by ellipsis in the `HistParent` table are gone. In a way the interval coalesce operation is similar to aggregation, but unlike aggregation, coalesce produces more than one aggregate value.

If we have a set of attributes identifying the child, then we could just define its lifetime the same way we defined the parent's. We don't have to, though! Instead of gluing the smaller [created, deleted] child intervals into the larger [birth, death], we just observe that if each individual [created, deleted] interval is contained in the parent lifetime, so also is the child lifetime.

Now everything is ready for formal constraint expression. The query

```
select * from HistChild c where not exists
  (select * from ParentLifetime p
   where p.id = c.pid
   and c.created between p.birth and p.death
   and c.deleted between p.birth and p.death
  )
```

enumerates all the child records that violate the temporal referential integrity constraint. Therefore, it should be empty.

Cardinality Constraint

The materialized view constraint enforcement method is like a hammer looking for a nail to strike. Cardinality constraints, that is, ensuring that a table has certain number of rows, surely fall into the nail category. Sometimes, there is a more ingenious solution.

Consider a table with 3 columns *A*, *B*, *C*, and functional dependency $\emptyset \rightarrow \{A, B, C\}$. In general, the functional dependency $x \rightarrow y$ requires each pair of rows that agree on the columns from the set *x* to agree on the columns from *y*, as well. In other words, there is no couple of rows such that they agree on column set *x* and disagree on *y*. In the case of functional dependency $\emptyset \rightarrow \{A, B, C\}$ this means no two rows (unconditionally) disagree on values of the columns *A*, *B*, *C*. These are the only columns in the table; therefore, all rows are identical! If we disallow duplicates (by enforcing unique key constraint), then we effectively enforced a constraint which limits table cardinality to 1, at most.

SQL lacks the ability to declare and enforce functional dependency constraints. A unique key is special case of functional dependency constraint $x \rightarrow y$, where *y* contains all table columns.

Unfortunately, unique keys with the empty set of columns are not allowed in SQL.

In one of the soap boxes in chapter 1 we had discussed a similar problem with `group by` operator that didn't admit empty sets either. As a workaround, we have introduced a (calculated) pseudo column. Let's amend the table with extra column⁸

```
table T (  
  A integer,  
  B integer,  
  C integer,  
  dummyCol integer default 0 not null check (dummyCol = 0) unique  
)
```

The combination of the check constraint and the uniqueness constraint guarantees that there is not allowed to be more than one row. Elegant solutions always trigger the same reaction: "Why didn't I think of that?"

Summary

- Materialized views provide the most comprehensive way to implement complex constraints.
- Constraints which are declared within RDBMS are superior to any alternative solution. Your query can be rewritten by optimizer to leverage constraint expression predicates.
- Be careful with concurrency issues when implementing constraints via triggers (or any other procedural way for that matter).

Exercises

1. Provide a set of symmetric functions of 4 variables.
2. What other operations besides `+`, `*`, `least`, and `greatest` can you suggest? What laws must these operations meet?
3. Write down the `ParentLifetime` view definition.

⁸ The solution is credited to Jarl Hermansson

4. Implement the `unique(n)` constraint, which permits no more than `n` rows to have any given value.
5. Implement a functional dependency constraint that restricts each `Emp.job` to a certain `Emp.sal` range. In other words, no two employees can have the same job, and have their salaries in different ranges. Use the expression `floor(log(10,Emp.sal))` from chapter 3 that maps salaries into logarithmic buckets.