Database Design with Identifying Surrogate Keys

Andrew Morgan

Analytical Logic cc

Johannesburg, South Africa

May 2000

Andrew.Morgan@telkomsa.net

Abstract

This paper describes surrogate keys and how they are used in database design. Surrogates are defined as efficient substitutes, and their purpose is to substitute themselves for large cumbersome keys. They are implemented as a native numeric type, usually integers.

Two types of surrogates are considered, identifying and non-identifying. Identifying designs implement the full composite key in details tables, whereas non-identifying designs only implement the simple key of its immediate parent. Although simpler to implement, non-identifying designs are massively disadvantaged because they are missing portions of the entire key, and consequently have problems like poor deep join performance.

Identifying designs are shown to be far superior. In the context of updating or resizing (conventional) primary keys, through substitution, surrogate key designs are unsurpassed in efficiency, flexibility and availability.

Table of Contents

Table of Illustrations

1 Introduction

Is it possible to design very robust and flexible databases, incorporating the controls of referential integrity without their inherent inflexibility? By inflexibility, it is implied the difficulties associated with making changes to the primary key of a table which is the target of a referential integrity constraint. I intend to show in this article that this can be achieved very successfully if a surrogate key design is used.

What are surrogate keys? The definition of surrogate is ***substitute*** or ***replacement***. I use the term in the ***substitute*** sense of the definition. That is, the primary key of a table is substituted by another smaller and more efficient key. The reasons for the substitution are the topic of this paper. Surrogate keys are substantially more flexible than conventional primary keys since they substitute the task of a conventional key without their usual limitations.

It is probably easiest to illustrate surrogate keys by example. Suppose one wished to execute the following design, represented in IDEF1X notation:

Illustration 1: High level schematic of a Timesheet system

A company has many regions, which have many departments, which have many employees, which have many timesheets.

The rest of the paper concerns itself with implementations of the above system using identifying conventional, non-identifying surrogate key and identifying surrogate key designs.

2 Conventional Design

In a conventional design, the relationships must be ***identifying*** (the parent primary key is migrated to the child to form a composite primary key). A typical implementation for the core section may look something like:

```
CREATE    TABLE     Company(
          comName   VARCHAR(32) NOT NULL,
          ...
          PRIMARY   KEY(comName));

CREATE    TABLE     Region(
          comName   VARCHAR(32) NOT NULL,
          regName   VARCHAR(32) NOT NULL,
          ...
          PRIMARY   KEY(comName,regName),
          FOREIGN   KEY(comName) REFERENCES Company);

CREATE    TABLE     Department(
          comName   VARCHAR(32) NOT NULL,
          regName   VARCHAR(32) NOT NULL,
          depName   VARCHAR(32) NOT NULL,
          ...
          PRIMARY   KEY(comName,regName,depName),
          FOREIGN   KEY(comName,regName) REFERENCES Region);

CREATE    TABLE     Employee(
          comName   VARCHAR(32) NOT NULL,
          regName   VARCHAR(32) NOT NULL,
          depName   VARCHAR(32) NOT NULL,
          empName   VARCHAR(32) NOT NULL,
          ...
          PRIMARY   KEY(comName,regName,depName,empName),
          FOREIGN   KEY(comName,regName,depName) REFERENCES Department);

CREATE    TABLE     Timesheet(
          comName   VARCHAR(32) NOT NULL,
          regName   VARCHAR(32) NOT NULL,
          depName   VARCHAR(32) NOT NULL,
          empName   VARCHAR(32) NOT NULL,
          timDate   DATE NOT NULL,
          ...
          PRIMARY   KEY(comName,regName,depName,empName,timDate),
          FOREIGN   KEY(comName,regName,depName,empName) REFERENCES Employee);
```

Notice that with an identifying scheme, it is necessary to make a composite primary key for all child tables. An identifying scheme is necessary so that the entire key chain is unambiguous and completely identified; hence the name.

2.1 Conventional Data

Company

comName
Borland
Corel

Region

comName	regName
Borland	Los Angeles
Borland	New York
Corel	Ontario
Corel	New York

Note: *regName* cannot uniquely identify the Region (two companies have New York Regions).

Department

comName	regName	depName
Borland	Los Angeles	IT
Borland	New York	IT
Corel	Ontario	IT
Corel	New York	IT

Note: *depName* cannot uniquely identify the department (all regions have IT departments). Neither can (*regName*,*depName*) uniquely identify a department (two companies have New York IT departments)!

Employee

comName	regName	depName	empName
Borland	Los Angeles	IT	Smith, AJ
Borland	New York	IT	Jones, BD
Corel	Ontario	IT	Black, MC
Corel	New York	IT	Brown, PR

Timesheet

comName	regName	depName	empName	timDate
Borland	Los Angeles	IT	Smith, AJ	2000/01/07
Borland	New York	IT	Jones, BD	2000/01/07
Corel	Ontario	IT	Black, MC	2000/01/07
Corel	New York	IT	Brown, PR	2000/01/07

Note: *timDate* cannot uniquely identify the time sheet (every employee has a time sheet for that date).

2.2 Substitution

The procedure now is to adapt the conventional design be implementing integer key substitutions and eventually derive the identifying surrogate key design.

2.2.1 Company

The primary key for *Company* is (*comName*). Let the integer *comID* represent *comName* and become the primary key of the table. It is necessary to retain *comName* to describe the row, but it is now an attribute of the primary key. The structure of the table becomes:

```
CREATE    TABLE     Company(
          comID     INTEGER NOT NULL,
          comName   VARCHAR(32) NOT NULL,
          ...
          PRIMARY   KEY(comID));
```

An alternate key (unique index) must be designated so that the new table is functionally equivalent to the old:

```
CREATE    UNIQUE INDEX COM_AK ON Company(comName);
```

Company is a top level table, and is simpler than the detail tables that follow.

2.2.2 Region

The primary key for *Region* is (*comName*, *regName*). Let the integer *regID* represent (*comName*, *regName*) and become the primary key of the table. It is necessary to retain *regName* to describe the row, but it is now an attribute of the primary key. In order for the referential integrity to *Company* be preserved, *comID* must also be present. The structure of the table becomes:

```
CREATE    TABLE     Region(
          regID     INTEGER NOT NULL,
          regName   VARCHAR(32) NOT NULL,
          comID     INTEGER NOT NULL
          ...
          PRIMARY   KEY(regID),
          FOREIGN   KEY(comID) REFERENCES Company);
```

An alternate key must be designated so that the new table is functionally equivalent to the old. Since the primary key was (*comName*,*regName*), and *comName* is represented by *comID*, the alternate key strategy is:

```
CREATE    UNIQUE INDEX REG_AK ON Region(comID,regName);
```

2.2.3 Department

The primary key for *Department* is (*comName, regName, depName*). Let the integer *depID* represent (*comName, regName, depName*) and become the primary key of the table. It is necessary to retain *depName* to describe the row, but it is now an attribute of the primary key. In order for the referential integrity to *Region* be preserved, *regID* must also be present. The structure of the table becomes:

```
CREATE    TABLE     Department(
          depID     INTEGER NOT NULL,
          depName   VARCHAR(32) NOT NULL,
          regID     INTEGER NOT NULL
          ...
          PRIMARY   KEY(depID),
          FOREIGN   KEY(regID) REFERENCES Region);
```

An alternate key must be designated so that the new table is functionally equivalent to the old. Since the primary key was (*comName, regName, depName)*, and (*comName, regName)* is represented by *regID*, the alternate key strategy is:

```
CREATE    UNIQUE INDEX DEP_AK ON Department(regID,depName);
```

2.2.4 Employee

The primary key for *Employee* is (*comName, regName, depName, empName*). Let the integer *empID* represent (*comName, regName, depName, empName*) and become the primary key of the table. It is necessary to retain *empName* to describe the row, but it is now an attribute of the primary key. In order for the referential integrity to *Department* be preserved, *depID* must also be present. The structure of the table becomes:

```
CREATE    TABLE     Employee(
          empID     INTEGER NOT NULL,
          empName   VARCHAR(32) NOT NULL,
          depID     INTEGER NOT NULL,
          ...
          PRIMARY   KEY(empID),
          FOREIGN   KEY(depID) REFERENCES Department);
```

An alternate key must be designated so that the new table is functionally equivalent to the old. Since the primary key was (*comName, regName, depName, empName*), and (*comName, regName, depName*) is represented by *depID*, the alternate key strategy is:

```
CREATE    UNIQUE INDEX EMP_AK ON Employee(depID,empName);
```

2.2.5 Timsheet

The primary key for *Timesheet* is (*comName, regName, depName, empName, timDate*). Let the integer *timID* represent (*comName, regName, depName, empName, timDate*) and become the primary key of the table. It is necessary to retain *timDate* to describe the row, but it is now an attribute of the primary key. In order for the referential integrity to *Employee* be preserved, *empID* must also be present. The structure of the table becomes:

```
CREATE    TABLE     Timesheet(
          timID     INTEGER NOT NULL,
          timDate   DATE NOT NULL,
          empID     INTEGER NOT NULL
          ...
          PRIMARY   KEY(timID),
          FOREIGN   KEY(empID) REFERENCES Employee);
```

An alternate key must be designated so that the new table is functionally equivalent to the old. Since the primary key was (*comName, regName, depName, empName, timDate*), and (*comName, regName, depName, empName*) is represented by *empID*, the alternate key strategy is:

```
CREATE    UNIQUE INDEX TIM_AK ON Timesheet(empID,timdate);
```

This completes the minimum substitution necessary for the two designs to be functionally equivalent insofar as they hold data. Note however, although (for example) *empID* represents (*comName, regName, depName, empName*), it is not possible to decompose it to recover the individual elements of the original key. Consequently, it is not possible to join *Timesheet* to *Company* or any of the tables higher in the chain except for *Employee*. The above process has derived a ***non-identifying surrogate key*** design. Non-identifying surrogate key designs are still some way short of being practically useful. They suffer from a number of debilitating problems (see section 4). Some additional fields are necessary for the surrogate key design to fully represent the original.

3 Identifying Surrogate Design

The task now at hand is to transform the non-identifying design into an identifying one. By examining the *Timesheet* table the solution can be seen, and is easily extrapolated backwards.

Recall that the original primary key of *Timesheet* was (*comName, regName, depName, empName, timDate*). In order to achieve an identifying design, it is necessary to be able to join from *Timesheet* to any table in the structure chain (*Company*, *Region*, *Department* or *Employee*).

Note also that by observing which surrogates represent what, the following is observed:

```
comName,   regName,   depName,   empName,   timDate

<-------------------- timID -------------------->

<--------------- empID ---------------->

<---------- depID ---------->

<----- regID ---->

<comID>
```

In order to satisfy the identifying nature of the original design, all surrogates of the original key must be present. This will not change the alternate key definition of the tables, but will change the primary keys. It is necessary to preserve the uniqueness of the individual keys (ie what were the primary keys in the non-identifying design), thus, a new index (the surrogate key index) takes on the role of the former primary key. The design becomes:

```
CREATE    TABLE     Company(
          comID     INTEGER NOT NULL,
          comName   VARCHAR(32) NOT NULL,
          ...
          PRIMARY   KEY(comID));
/*
Omitted because it would be the same as the primary key.
CREATE    UNIQUE    INDEX COM_SK ON Company(comID)
*/
CREATE    UNIQUE    INDEX COM_AK ON Company(comName);

CREATE    TABLE     Region(
          comID     INTEGER NOT NULL,
          regID     INTEGER NOT NULL,
          regName   VARCHAR(32) NOT NULL,
          ...
          PRIMARY   KEY(comID,regID),
          FOREIGN   KEY(comID) REFERENCES Company);
CREATE    UNIQUE    INDEX REG_SK ON Region(regID)
CREATE    UNIQUE    INDEX REG_AK ON Region(comID,regName);
```

```
CREATE    TABLE     Department(
          comID     INTEGER NOT NULL,
          regID     INTEGER NOT NULL,
          depID     INTEGER NOT NULL,
          depName   VARCHAR(32) NOT NULL,
          ...
          PRIMARY   KEY(comID,regID,depID),
          FOREIGN   KEY(comID,regID) REFERENCES Region);
CREATE    UNIQUE    INDEX DEP_SK ON Department(depID);
CREATE    UNIQUE    INDEX DEP_AK ON Department(regID,depName);

CREATE    TABLE     Employee(
          comID     INTEGER NOT NULL,
          regID     INTEGER NOT NULL,
          depID     INTEGER NOT NULL,
          empID     INTEGER NOT NULL,
          empName   VARCHAR(32) NOT NULL,
          ...
          PRIMARY   KEY(comID,regID,empID,depID),
          FOREIGN   KEY(comID,regID,depID) REFERENCES Department);
CREATE    UNIQUE    INDEX EMP_SK ON Employee(empID);
CREATE    UNIQUE    INDEX EMP_AK ON Employee(depID,empName);

CREATE    TABLE     Timesheet(
          comID     INTEGER NOT NULL,
          regID     INTEGER NOT NULL,
          depID     INTEGER NOT NULL,
          empID     INTEGER NOT NULL,
          timID     INTEGER NOT NULL,
          timDate   DATE NOT NULL,
          ...
          PRIMARY   KEY(comID,regID,depID,empID,timID),
          FOREIGN   KEY(comID,regID,depID,empID) REFERENCES Employee);
CREATE    UNIQUE    INDEX TIM_SK ON Timesheet(timID);
CREATE    UNIQUE    INDEX TIM_AK ON Timesheet(empID,timDate);
```

One of the great advantages of identifying surrogate key designs this that the primary key becomes an internal mechanism of the database, whilst the alternate keys implement the conventional user interface (the surrogate keys *xxxID* are never exposed to an end user, whilst the interface fields *xxxName* are). By ***decoupling*** these tasks a number of interesting possibilities arise. Recall that the alternate key strategy on *Employee* is (*depID*, *empName*). Both designs (conventional and surrogate) allow a particular employee to reside in multiple departments. It is now possible to tighten the model to say (*regID,empName*) or even (*comID, empName*) depending on the system being modelled. This does not effect the primary keys in any way, and the decision may be deferred or even changed after the system is up and running! Assuming no data violates the condition, the following could be done:

```
DROP      INDEX     EMP_AK;
CREATE    UNIQUE    INDEX EMP_AK ON Employee(regID,empName);
```

This change in strategy has no effect on the front-end application at all. The ability to defer or change the alternate key strategy is a flexibility the conventional design simply

cannot achieve! This is because the conventional design couples the referential integrity mechanism with the interface into the primary key definitions which cannot be changed (easily) after construction.

Note that the composite primary keys are ordered top-down (comID..xxxID). The reason for this is that because there is a surrogate key (xxxID) for all detail tables, the primary key index has a different selectivity.

3.1 Surrogate Data

Company

comID	comName
1	Borland
2	Corel

Note: *comName* is unique.

Region

comID	regID	regName
1	11	Los Angeles
1	12	New York
2	13	Ontario
2	14	New York

Note: *regID* uniquely identifies the region (even though two have the same name). Also (*comID*, *regName*) is unique.

Department

comID	regID	depID	DepName
1	11	21	IT
1	12	22	IT
2	13	23	IT
2	14	24	IT

Note: *depID* uniquely identifies the department (even though four have the same name). Also (*regID*, *depName*) is unique.

Employee

comID	regID	depID	empID	empName
1	11	21	31	Smith, AJ
1	12	22	32	Jones, BD
2	13	23	33	Black, MC
2	14	24	34	Brown, PR

Note: (*depID*, *empName*) is unique.

Timesheet

comID	regID	depID	empID	timID	timDate
1	11	21	31	41	2000/01/07
1	12	22	32	42	2000/01/07
2	13	23	33	43	2000/01/07
2	14	24	34	44	2000/01/07

Note: *timID* uniquely identifies the timesheet (even though four have the same date). Also (*empID*, *timDate*) is unique.

Notice that the actual values of the surrogates are irrelevant (as long as they are consistent). In the above data samples, the keys have been started from different series for clarity. In reality, each sequence would typically start at 1 and using generators to produce them is very effective.

The default alternate key strategy it starting to emerge by examining the shaded columns in the tables above. It normally consists of the surrogate key of the immediate parent table (the primary key if the parent is a top level table), coupled with the last component of the conventional key.

There is a great deal of meaning wrapped up in these surrogate keys:

Key Value	Key Chain	Represents
comID = 1	[1]	Borland
regID = 12	1[,12]	Borland / New York
depID = 23	2,13[,23]	Corel / Ontario / IT
empID = 34	2,14,24[,34]	Corel / New York / IT / PR Brown
timID = 41	1,11,21,31[,41]	Borland / Los Angeles / IT / AJ Smith / 2000/01/07

4 Problems with Non-identifying Surrogate Key Designs4

Non-identifying surrogate key designs are still some way short of being practically useful. They suffer from a number of debilitating problems.

4.1 Poor Deep Joins

Supposing one needs to join the *Timesheet* and *Company* tables. This must be done as follows:

```
...
FROM      Timesheet TIM
JOIN      Employee EMP
          ON        TIM.empID = EMP.empID
JOIN      Department DEP
          ON        EMP.depID = DEP.depID
JOIN      Region REG
          ON        DEP.regID = REG.regID
JOIN      Company COM
          ON        REG.comID = COM.comID
...
```

Since the key structure is non-identifying, the key chain is incomplete, and each intermediate table must be joined to find the target table.

The identifying equivalent is:

```
...
FROM      Timesheet TIM
JOIN      Company COM
          ON        TIM.comID = COM.comID
...
```

because all elements of the key chain are present and intermediate tables in the chain may be jumped.

4.1.1 No Key Unification

Key unification results from an identifying key structure where one table references another table and portions of their respective primary keys have common elements. The common elements are unified so that there are not multiple copies of those fields in the table.

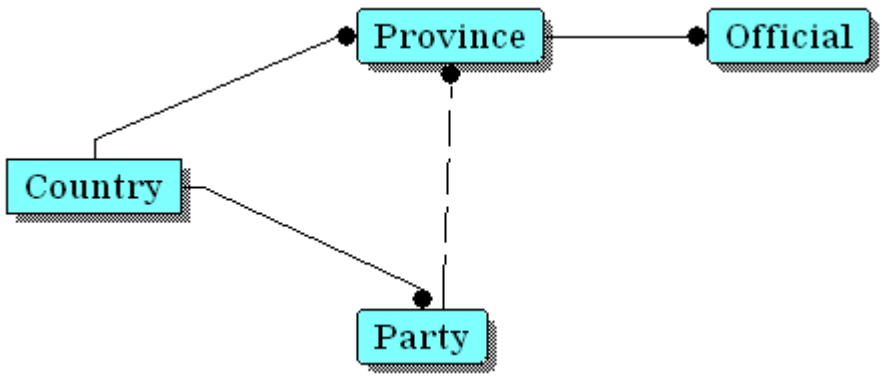

Illustration 2: High level schematic of a Political system

This design shows a country with many provinces and many political parties. A province has many officials. The relationship from *Party* to *Province* indicates which political party governs that province.

In a non-identifying design, the primary keys for *Country*, *Province*, *Party* and *Official* are respectively (*couID*), (*proID*), (*parID*) and (*offID*). The only restriction on the *Province* / *Party* relationship is that it is valid.

In an identifying design, the primary keys for *Country*, *Province*, *Party* and *Official* are respectively (*couID*), (*couID*,*proID*), (*couID*,*parID*) and (*couID*,*proID*,*offID*), and key chains are listed in the same order. *Province* and *Party* both have *couID* in common, and so they unify (that is, there is only one *couID* field in *Province*). Consequently it is a foreign key violation to attempt to set the governing party of a province to a party from another country!

Consider the following data:

Country

couID	couName
1	United States
2	South Africa

Party

couID	parID	parName
1	1	Democrats
1	2	Republicans
2	3	Democratic Alliance
2	4	African National Congress

Province

couID	proID	parID	proName
1	1	1	California
1	2	2	New York
1	3	3	Ohio
2	4	4	Western Cape
2	5	5	Gauteng

Official

couID	proID	offID	offName
1	1	1	Arnold Schwarzenegger
1	1	2	
1	2	3	Guilane
1	3	4	?
2	4	5	?
2	5	6	Tshilowe

If we now want to update the *Province* table, and set the ANC (from South Africa) as the governing party of California, there are two ways to go about it:

Update *parID* in *Province*

```
UPDATE    Province SET
          parID = 4
WHERE     proID = 1
```

However, because the *Country* in *Province* is 1, the attempted foreign key chain (1,4) is not valid in *Party* (it is (2,4)) and a foreign key violation results.

Update *couID* and *parID* in *Province*

```
UPDATE    Province SET
          couID = 2,
          parID = 4
WHERE     proID = 1
```

This gets passed the above problem by effectively trying to move the province into a different country. This update attempts to change the primary key chain (1,1) in *Province* to (2,1). But this update would invalidate the the primary key chain (1,1,1) for Arnold Schwarzenegger in *Official*, and again a foreign key violation results.

Note that in a non-identifying design this update would be allowed because the *Province* table does not know which country the referenced *Party* belongs to.

Key unification is an extremely beneficial bonus of an identifying structure. It is almost a free service to enforce a consistent design. Non-identifying schemes should be avoided.

5 Size Comparison

At this point it is worth noting how these implementations are different. Notice that their content is identical although not specified (... in the source listings) except for the presence of surrogate keys and coupled alternate keys in the surrogate design. In the tables below, ***IC*** is an ***identifying conventional***, and ***IS*** is an ***identifying surrogate***. All sizes ignore the unidentified portions of the design.

5.1 Size (bytes per row)

Table	IC	IS	IS/IC %
Company	32	36	112.5
Region	64	40	62.5
Department	96	44	45.8
Employee	128	48	37.5
Timesheet	132	20	15.2

Surrogate tables are always smaller from the first detail table down and become increasingly smaller the deeper the structure.

5.2 Primary Keys (bytes per row)

Table	IC	IS	IS/IC %
Company	32	4	12.5
Region	64	8	12.5
Department	96	12	12.5
Employee	128	16	12.5
Timesheet	132	20	15.2

Surrogate tables always have smaller primary keys, usually a specific fraction of a conventional key size (because typically a series of VARCHAR fields have been mapped to a series of INTEGER fields).

5.3 Foreign Keys (bytes per row)

Table	IC	IS	IS/IC %
Region	32	4	12.5
Department	64	8	12.5
Employee	96	12	12.5
Timesheet	128	16	12.5

Surrogate tables always have smaller foreign keys, usually a specific fraction of a conventional key size.

5.4 Coupled Alternate and Surrogate Keys (bytes per row)

Table	IT	IS (AK)	IS (SK)
Company	0	32	4
Region	0	36	4
Department	0	36	4
Employee	0	36	4
Timesheet	0	8	4

Surrogate tables always require an additional alternate and surrogate key, however, their sizes are usually fixed and generally do not increase as the structure deepens.

Notice that all the surrogate primary and foreign keys are substantially smaller and are massively efficient (primarily integer based) compared to the conventional design! Also, the losses incurred by the necessary inclusion of composite surrogate and alternate keys are more than recovered by the exclusion of huge composite primary and foreign keys.

5.5 Disadvantages

One disadvantage is that tables are limited to 2^{32} (approx. 4,200,000,000) rows (for 32-bit integers). Some databases are already supporting 64-bit integers, in which case, the tables are limited to 2^{64} (approx. 18,400,000,000,000,000,000) rows, which is much less of a problem.

Another is a rather peculiar behaviour which surrogate designs exhibit by virtue of the fact that keys are mapped to integers. It is legal (although ***meaningless***) to join arbitrary surrogates together. For example:

```
...
FROM      Company COM
JOIN      Region REG
          ON        REG.regID = COM.comID
...
```

This is analogous to:

```
...
FROM      Company COM
JOIN      Region REG
          ON        REG.regName = COM.comName
...
```

The difference is that *regName* will almost never match *comName* whereas *regID* will most likely match some range of *comID* because they are both numeric sequences originating at one. As a general rule, the join will succeed by the table of lower cardinality. The fault essentially lies with the fact that SQL implements very poor (if any) type checking.

The best long-term solution is to create a different domain (data type) for each surrogate. The emergence of proper object / relational database management systems in the future [1] would question invalid (type mismatched) joins, if not reject them outright. Thus, the surrogate join described above would simply be illegal. In fact, the only legal joins would be the explicit foreign keys identified in the design that were correctly migrated.

In the short term, different domains may achieve little, since some SQL engines will even allow:

```
...
FROM      Company COM
JOIN      Region REG
          ON        REG.regID = COM.comName
...
```

This is crazy, since the engine is performing implicit type casting which is not within the scope of a pure relational database (relational theory requires explicit typecasts [2]). Since this is the case, joins (especially surrogate) must be done very carefully.

5.6 Advantages

This is where the surrogate key design really comes into its own. It is very effective at selecting data from joined tables with minimal join conditions and is exceptionally adaptable to data and structure changes.

5.6.1 Minimal Joins

To demonstrate this, we will make a modification to the design as follows:

Illustration 3: High level schematic of an upgraded Timesheet system

The modifications include a backward link from employee to department to indicate the head of department. We now wish to sum hours by the head of the department's budget (*empBudget*); admittedly a contrived example!

Without specifying the modified design (except to note that Department has a rolenamed foreign key *empNameHead*), here is the select:

```
SELECT    empBudget,SUM(timHours)
FROM      Timesheet TIM
--        Find the head of the department.
JOIN      Department DEP
          ON        TIM.comName = DEP.comName
          AND       TIM.regName = DEP.regName
          AND       TIM.depName = DEP.depName
--        Find the head's budget
JOIN      Employee EMP
          ON        DEP.comName = EMP.comName
          AND       DEP.regName = EMP.regName
          AND       DEP.depName = EMP.depName
          AND       DEP.empNameHead = EMP.empName
GROUP     BY empBudget
```

The identifying surrogate key equivalent (with Department holding a rolenamed foreign key *empHID*) is:

```
SELECT    empBudget,SUM(timHours)
FROM      Timesheet TIM
JOIN      Department DEP
          ON        TIM.depID = DEP.depID
JOIN      Employee EMP
          ON        DEP.empHID = EMP.empID
GROUP     BY empBudget
```

The difference is that because each surrogate key is unique in its own right, it is not necessary to join with the whole key chain.

5.6.2 Primary Key Data and Structure Changes

Let us say that one wanted to update *comName* from "Borland" to "Inprise". Unless the conventional system supports cascading updates, referential integrity prevents this action because the name "Borland" appears as a portion of foreign keys all over the system. Even with cascading updates, this is a kludge, because the database must update ***every*** record that refers to "Borland". This could be millions of rows! To accomplish this manually, ***all*** foreign keys must be dropped, ***all*** fields of ***all*** effected tables must be updated and then the foreign keys reapplied. This requires structural and data changes, and most likely taking the system offline.

In the surrogate design, updating *comName* (on a live system) has absolutely no effect on the rest of the system whatsoever, since all foreign key references are through its substitute (surrogate). This can be done as follows:

```
UPDATE    Company   SET
          comName = 'Inprise'
WHERE     comName = 'Borland';
```

Assuming that you can live with cascading updates. The next problem is even worse. Let us say that VarChar(32) is no longer big enough, and one wanted to change it to 64. Unless the conventional system allows cascading structure changes, referential integrity prevents this action because all instances of the foreign key must be the same type. To accomplish this manually, ***all*** foreign keys must be dropped, ***all*** reference fields of ***all*** affected tables must have their type changed and then the foreign keys must be reapplied. This requires structural and data changes, and the system will have to be taken offline.

Again, in the surrogate design, changing *comName's* size (on a live system) has absolutely no effect on the rest of the system whatsoever, since all foreign key references are through its substitute (surrogate). This can be done as follows:

```
ALTER     TABLE     Company ADD temp VARCHAR(32);
UPDATE    Company   SET temp = comName;
DROP      INDEX     COM_AK;
ALTER     TABLE     Company DROP COLUMN comName;
ALTER     TABLE     Company ADD comName VARCHAR(64);
UPDATE    Company   SET comName = temp;
CREATE    UNIQUE    INDEX COM_AK ON Company(comName);
ALTER     TABLE     Company DROP COLUMN temp;
```

Although this action requires structural changes, they are only to a ***single*** index and a ***single*** table!

The renaming of "Borland" to "Inprise" and changing the field to VARCHAR(64) could all be achieved within about five minutes with virtually no down time! The system can remain online as long as non-select operations on *Company* are blocked.

The very important point here, is that none of the above changes are unreasonable! We regularly get requests to change drawing, document or package numbers (either through a legitimate mistake or a reorganization). If these numbers are directly linked to transmittals, time sheets and the like, changing them is very difficult. In our experience, these types of changes are remarkably common, and have cost us massive amounts of downtime in the past using conventional designs. With the surrogate key technique, our database designs are:

- Smaller
- More efficient
- Usually faster
- Extremely adaptable to change
- Experience minimal downtime

5.7 Normalization

It has been suggested that surrogate key designs violate normalization, however, this simply is not true. Both the surrogate key, and the conventional key are ***candidate*** keys of the table. Higher form normalization rules state that Functional Dependencies (FD's - BCNF), Multi-Value Dependencies (MVD's - 4NF) and Join Dependencies (JD's - 5NF) must be implied by the ***candidate keys*** of a table, not the primary key ***per se***.

6 Conclusion

Surrogates are unfortunately one of those database facets which tends to be highly polarized -- you either love or hate them. The benefits I have achieved by their use so far outweigh their disadvantages that I would probably never contemplate another non-surrogate design[1].

It is possible to design extremely flexible and adaptable systems by employing surrogate keys, that satisfy normalization rules. The advantages of such a design enormously outweigh their slight overhead (which can be rapidly recovered anyway).

Change Type	Conventional	Surrogate
Update PK	Drop all FK's Update all PK references Reapply all FK's	Update "Key"
Change PK type	Drop all FK's Change all FK sizes Update all PK references Reapply all FK's	Drop a single AK Change "key" size Reapply a single AK

In a conventional design, these tasks require ***structural*** changes in both cases, and the second requires structural changes against ***every*** table that references the changing key. Apart from the structural change to a ***single*** table and a ***single*** index, the surrogate design accomplishes these tasks by data updates alone! These properties make surrogate designs extremely efficient, compact, flexible, adaptable and available.

1 In the five years since writing this article, this has proven true!

Bibliography

1 Chris Date Foundation for Object/Relational Databases (2000)
2 Chris Date An Introduction to Database Systems (2000)