Database De-Centralization — A Practical Approach

Tor Didriksen
SINTEF DELAB, Trondheim, Norway
Tor.Didriksen@delab.sintef.no

César Galindo-Legaria
SINTEF DELAB, Trondheim, Norway
cesar@acm.org

Eirik Dahle
Telenor Research, Kjeller, Norway
Eirik.Dahle@tf.tele.no

Abstract

We describe a scheme to fragment and distribute centralized databases. The problem is motivated by trends towards down-sizing and reorganization, reflecting actual, often distributed responsibilities within companies. A major practical requirement is that existing application code must be left unchanged.
We present SQL extensions to specify ownership and data replication information declaratively. From this, a compiler generates triggers and view definitions that implement the distributed scheme, on top of a collection of local databases. Our strategy has been applied successfully at Telenor — the Norwegian telephone company.

1 Introduction

It is known that distributed databases are often more cost-effective than their centralized counterparts, reflect better the organizational structure of companies, allow incremental growth, and have the potential of increasing performance, reliability, and availability of database systems [CP85, OV91]. The replacement of centralized systems by distributed solutions is clearly a trend, driven not by the availability of new equipment but rather by profound restructuring changes in the way companies operate. Many companies however, already have a huge investment in centralized database systems, and the prohibitive cost of re-implementing their current, working systems prevents them from adopting distributed solutions that would fit their needs better. Instead, what they need is an *evolutionary database de-centralization* path.

**Proceedings of the 21st VLDB Conference
Zurich, Switzerland, 1995**

In a sense, the database de-centralization is the converse of *database federation*, which aim at presenting an integrated system on top of independently developed databases [SL90]. Both re-engineering mechanisms rely heavily on concepts and techniques developed for *multi-databases* [LMR90] but, although some concerns are common, each has its particular needs. Implementable solutions to federation and de-centralization are likely to be adopted relatively quickly in practice, as they address the day-to-day concerns of dynamic, evolving companies.

Database de-centralization consists of replacing a single, central data repository by a collection of interconnected *local* databases, without restricting the services formerly provided by the central system. Technically, the distributed model has the potential of improving performance and data availability, as local databases are likely to be smaller, locking contention can be reduced, transaction load can be split over several processors, and data can be placed closer to distributed users. Administratively, it can also shift a share of the responsibility over both data and hardware to the actual producers/consumers of information. In addition to the typical issues of distributed database design, de-centralization introduces the notion of data ownership.

In many cases, the obvious alternative of replacing the centralized database by a sophisticated distributed database (e.g. with two-phase commit, distributed query processing, voting mechanisms for replica consistency, and so on) is impractical. Some features of distributed databases are not always needed for de-centralization, yet they come at a cost and introduce extra overhead; and, perhaps more importantly, the interface offered to applications may change for the distributed database (e.g. we might be forced to change database vendors), thus making necessary a thorough revision of all existing applications.

The approach developed in this paper is based on having a collection of local databases communicating in a loosely coupled fashion, in the style described in [GMK88]. The DBMS formerly used for the central-

ized system is then used for each local database, with an additional thin "distribution layer" implemented with triggers. Our work includes SQL extensions for the specification of data fragmentation and distribution, from which a compiler derives the necessary triggers. This approach requires no modifications to existing applications, the existing database schema, or underlying DBMS, and can be implemented with off-the-shelf components. Clearly, giving up mechanisms such as two-phase commit affects the behavior and properties of the distributed system. Although our specific solution is suitable only for applications that can tolerate slightly out-of-date data from remote machines, we do point out issues that arise in any data de-centralization process.

This paper is organized as follows. The remainder of this section describes the DIBAS database de-centralization project. Section 2 presents an overview of our distributed architecture. Section 3 introduces our formal model. Section 4 and 5 deal with the specification of data ownership and distribution, respectively. Section 6 summarizes our proposal and compares it with other alternatives. Section 7 presents our conclusions. Examples of fragmentation rules, and generated SQL code are found in the appendix.

1.1 The DIBAS project

DIBAS (DIstributed dataBASe) is a joint project of SINTEF DELAB with the research division of Telenor (a state-owned Norwegian Telecom company) to improve the efficiency of some of the databases and information systems of the operational divisions. Like other providers of telephone services in Europe, Telenor is being pushed to increase competitivity as a result of governmental liberalization policies. For the database infrastructure and services, a driving concern is to have flexibility in the assignment of responsibilities and costs, in a climate of administrative restructuring, with short-term cost-effective solutions.

We present here the principles used to restructure one of the databases of the operational division of Telenor. The database contains over 150 tables with information about utilization of telephone/isdn trunk lines and switches, and changes projected for the future. There is a mix of analog and digital equipment, and both historical data, current status, prognosis, and plans are maintained. Applications on the database include network configuration, traffic measurement and statistics, routing, subscriber usage, prognosis and planning.

2 Architectural overview

In addition to local autonomy, a prime motivation for database de-centralization is the potential of improving performance and data availability: Each local database will be (much) smaller than a centralized database, locking contention and network traffic can be reduced, and the transaction load can be split over several processors. A major requirement in our case is that the database must be *partially* replicated, that replication is *transparent* to the users (applications and programmers), and that data must be replicated in a *peer-to-peer* (as opposed to "master-slave") fashion. Other requirements to a possible de-centralization of the database were that no modifications could be done to existing applications and their databases or to the underlying DBMS.

Briefly, the basic idea of DIBAS is to take a centralized implementation, determine data ownership, and store data at each owner site. Data is replicated according to user and application needs, so that all data access can be done locally at each site. Conflicting updates are avoided by ensuring that data imported from (owned by) other sites is read-only. Distribution of replicated data is done incrementally and asynchronously. The scheme of a site with a local database is shown in Figure 1, whose components are explained below.

2.1 Fragmentation and data ownership

A key concept for de-centralization is data *ownership*, which induces a partitioning of the centralized database. Each table of the database is partitioned into horizontal non-overlapping fragments, and an owner is assigned to each fragment. Ownership entails exclusive write access, as well as control over who else can read specific portions of data.

Although this can be generalized, we assume that owners are identified with sites and local databases. The assumption is consistent with common working environments in corporations and, frequently this data fragmentation directly reflects responsibility for real-world objects described by the data. This also suggests a natural starting point for data distribution: Each site owns a fragment of every table of the original database, and each fragment is stored in the local database of its owner. Fragments may be empty, contain some, or all the tuples of a table.

2.2 Data replication

Users do not issue distributed transactions in our architecture. Instead, transactions are submitted to local databases. Since only local data is available, to access data owned by other sites, it must be *imported* beforehand. Data owners specify the portion of their data that they are willing to *export* to other sites. The process of export/import implements a form of asynchronous data replication, which is maintained by the

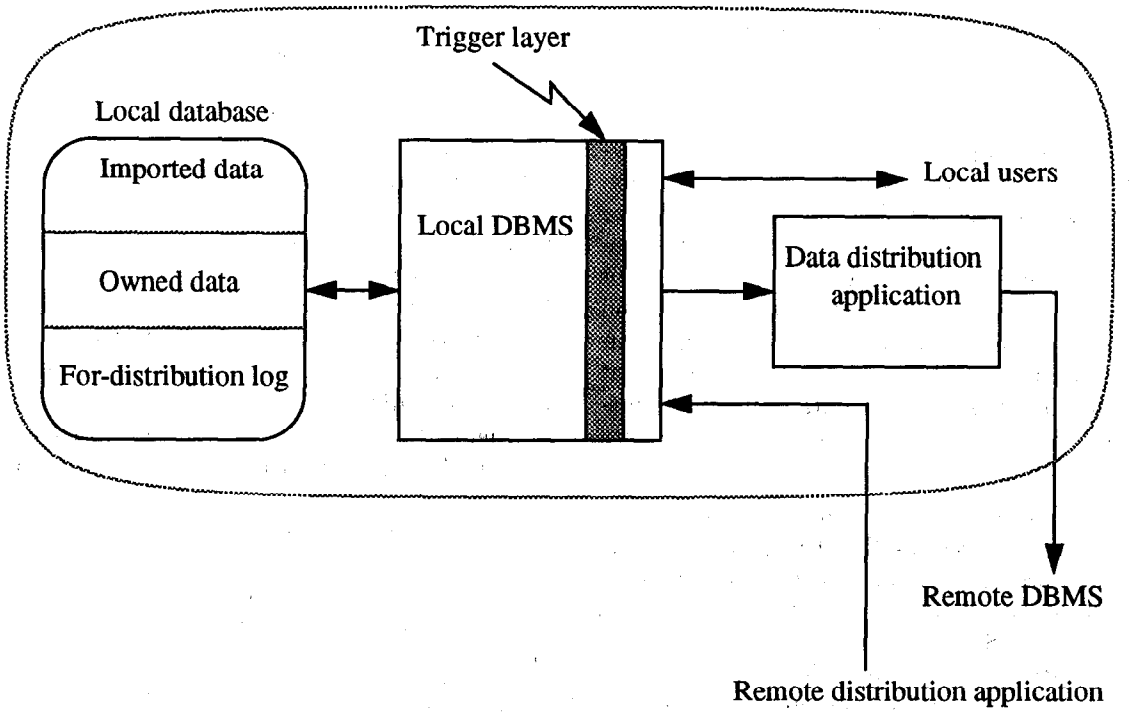

Figure 1: Site with local database.

system. Imported and owned fragments of a given table are stored in the same physical table of the local database. This allows efficient processing of read-only transactions, but can lead to inconsistencies for read-write transactions. Therefore, imported data can be read only.

Restricting imported data to be read-only does not guarantee global serializability, but there is still a range of correctness criteria applicable, depending on the choice of a data propagation policy [GMK88]. Techniques for the asynchronous update of replicated data include gossip protocols [HHW89], submission of transactions on persistent queues [CW93], and periodical or on-demand distributed update transactions. We use the latter approach: A data distribution application replicates data using distributed transactions, and two-phase commit.

2.3 Triggers

To enforce ownership constraints and to support incremental update of replicated data, we rely on the use of *triggers*, currently available in most commercial DBMSs [Sto92]. Such triggers are automatically activated by the DBMS in response to data manipulation requests issued by transactions. The specifics of when and how triggers are activated differ in various systems and theoretical models [MD89, SJGP90, WCL91, Syb90]. But they provide an architecturally sound place to insert a layer of additional services between applications and the DBMS, as is done in [CW90, CW91, CW93].

Like [CW90, CW91, CW93], we take a list of declarative specifications and compile them into a collection of database triggers, which implement the required service. In our case, it is ownership and replication rules that are declaratively specified. The compiler produces triggers, view definitions, and stored procedures that are installed in the local databases. These triggers verify ownership and support data replication, as explained below.

For concreteness, we assume Sybase triggers [Syb90], which behave as follows. A trigger is a precompiled stored procedure which can contain an arbitrary sequence of SQL statements, and is associated with a pair of the form [*table*, *operation*], where *operation* is either *insert*, *delete*, or *update*. A trigger on $[R, o]$ is executed automatically after an SQL statement performs operation o on table R on behalf of some transaction, and before the next SQL statement of the same transaction. At most one trigger is allowed per pair $[R, o]$. The SQL statements of the trigger body may refer to any part of the database, and has special access to the data modified by the triggering operation: Old values of modified tuples are available in the `deleted` virtual table, new tuple values are available in the `inserted` virtual table. Sybase trigger firing and execution is thus *set-oriented* [HW93]. Triggers fire on simple modification events, but arbitrary database transition conditions may be specified using the `deleted` and `inserted` tables. A trigger which

modifies the database state, may fire other triggers recursively.

2.4 Incremental update of replicas

Owner sites keep track of the data they have exported to other sites to update replicated data in an incremental way. Instead of re-sending complete table fragments, they send only the necessary deletions and insertions to be executed at the remote site. To accomplish this, triggers automatically store modifications to locally owned data, with timestamp and operation, in *for-distribution* log tables. We maintain one *for-distribution* log table for each replicated base table.

A separate data-distribution application performs asynchronous propagation of replicated data from the *for-distribution* log tables, taking into account the timestamp of updates previously sent to each site. Local timestamps are adequate for this incremental replication scheme. To ensure the consistency of incremental updates, we need the guarantee that data propagation commands are committed at the remote sites.

3 Formal model

In this section we describe a formal model behind our approach, and discuss correctness guarantees that can be given by the system.

3.1 Ownership and distribution predicates

An important property of our system is that tuple ownership is not always defined solely on the content of an individual tuple (e. g. attribute value in some specific range), but may also depend on the the database state. For example, ownership of a tuple with an external key may be defined to be the same as that of the tuple referenced. *Ownership* is specified on a table-by-table basis, using a predicate with three arguments: $\mathcal{O}_R(t, \mathcal{DB}, S)$; such predicate returns TRUE if tuple t, consistent with the schema of R, is owned by site S, given database contents DB. It corresponds, in practice, to an SQL subquery with free variables t, S, returning an empty/non-empty result. For correctness, ownership predicates must assign *a unique owner* to each tuple. That is, the ownership must partition the database.

Distribution is handled similarly. However, to allow selective export of data to different sites, the predicate uses both source and destination site: $\mathcal{D}_R(t, \mathcal{DB}, S_s, S_d)$. We allow multiple, independent distribution predicates for each table. In practice, the distribution predicate for a table corresponds to a union of SQL queries, returning the set of tuples to be replicated from a sender site to a recipient site.

3.2 Database fragmentation and locality

The central database is fragmented into a collection of databases, based on the predicates defined above. Given an ownership predicate $\mathcal{O}_R$ for relation R, the *fragment* $\mathcal{F}(\mathcal{O}_R, \mathcal{DB}, S) \subseteq R$ consists of those tuples in R owned by site S. For a collection of ownership predicates $\mathcal{O} = \mathcal{O}_{R_1}, \ldots, \mathcal{O}_{R_n}$, the *database fragment* owned by S is $\mathcal{F}(\mathcal{O}, \mathcal{DB}, S)$. Fragments based on distribution predicates are denoted similarly.

In our architecture, the central database DB is replaced by a collection of local databases $\mathcal{DB}_{S_1}, \ldots, \mathcal{DB}_{S_m}$. Given a collection of ownership and distribution predicates, $\mathcal{O}, \mathcal{D}$, each local database consists of the data it owns, plus the data it imports from other sites:

$$\mathcal{DB}_{S_i} = \mathcal{F}(\mathcal{O}, \mathcal{DB}, S_i) \cup \bigcup_{j \neq i} \mathcal{F}(\mathcal{D}, \mathcal{DB}, S_j, S_i).$$

The above expression amounts to the initial distribution of the centralized data to the different local databases. After distribution, *locality* is a basic assumption of our approach. In particular, each site must be able to determine, based only on its local data, what it owns and what it must export. Formally, this becomes: For every S_i, $\mathcal{F}(\mathcal{O}, \mathcal{DB}, S_i) = \mathcal{F}(\mathcal{O}, \mathcal{DB}_{S_i}, S_i)$; and for every $S_j \neq S_i$, $\mathcal{F}(\mathcal{D}, \mathcal{DB}, S_i, S_j) = \mathcal{F}(\mathcal{D}, \mathcal{DB}_{S_i}, S_i, S_j)$.

A way to guarantee locality is to make sure data dependencies "do not cross ownership boundaries." That is, if ownership of tuple t_1 depends on that of t_2, then their owner is the same. In this case, we have a fixed-point

$$\mathcal{F}(\mathcal{O}, \mathcal{DB}, S_i) = \mathcal{F}(\mathcal{O}, \mathcal{F}(\mathcal{O}, \mathcal{DB}, S_i), S_i).$$

There is a similar property for distribution predicates. Sometimes, it is necessary for dependencies to cross ownership boundaries. However, our current experience indicates that this kind of dependency is rare, and can be handled as follows. Extra data $\mathcal{DB}_0$ needed to determine ownership should be distributed to all sites, and $\mathcal{DB}_0$ itself should never be modified during normal operation. Modifying $\mathcal{DB}_0$ would be handled as database reconfigurations, by special protocols. Relations in $\mathcal{DB}_0$ can be regarded as *dictionary* tables, are owned by a central site, and are replicated in their entirety to all other sites.

Locality is also critical to continue running formerly centralized applications. An application can be executed correctly at site S_i, if all the data it requires is contained in $\mathcal{DB}_{S_i}$, and every tuple t it deletes or inserts (modeling updates as deletion followed by insertion) is owned by S_i.

3.3 Convergence of triggers

Since our architecture relies on the use of triggers, we need to address the issue of convergence of those triggers [ZH90, AWH92].

With respect to ownership, triggers must ensure that read-only fragments imported from other sites are not modified by local transactions. If data ownership can be tested locally, then triggers can verify appropriate ownership of every modified tuple and abort transactions that attempt to violate this constraint.

When, as a result of a local transaction in S_i, a distribution fragment $\mathcal{F}(\mathcal{D}, \mathcal{DB}_{S_i}, S_i, S_j)$ changes, the modifications must be sent to and installed in the database of S_j. We say that the *data propagation is pending* until the modifications are actually sent and installed at the remote site. Locality can be used to guarantee that if local transactions stop being submitted, then in finite time trigger activation will also stop, and data will be replicated correctly. Assume that distribution can be determined locally, by owned data; that is, for every $S_i \neq S_j$,

$$\mathcal{F}(\mathcal{D}, \mathcal{DB}, S_i, S_j) = \mathcal{F}(\mathcal{D}, \mathcal{F}(\mathcal{O}, \mathcal{DB}, S_i), S_i, S_j).$$

Data distributed to some site S_j cannot affect the data owned by S_j, and therefore no further data distribution is initiated.

3.4 Serializability of transactions

Asynchronous propagation of replicated data gives up global serializability of transactions in favor of greater availability and performance. Local transactions have no guarantee that their imported data is up to date (although users should be able to request the system for a "data re-fresh").

The notion of *fragmentwise serializability* defined in [GMK88] applies to our architecture as follows: (1) Schedules consisting solely of transactions issued on a given site S_i are serializable; and (2) data distribution can be done so replicated data is atomically updated, and reflects no partial effects of transactions committed at other sites.

4 Ownership types

We distinguish several cases occurring in practice where the specification of ownership can be made in a simple fashion, and appropriate ownership tests can be generated automatically. These special cases are identified as *ownership types*. Based on these ownership types we generate triggers which reject transactions that attempt to modify data not owned by the site, and log valid changes in a *for-distribution* log table. The generated triggers have the following form:[1]

```
if exists (
  select * from deleted   MODIFIED[2]
  where not ( <ownership predicate> ) )
or exists (
  select * from inserted  MODIFIED
  where not ( <ownership predicate> ) )
then begin
  rollback transaction
  return
end
else begin /* log changes */
  insert into <for-distribution log>
  select getdate(), 'd' , * from deleted
  insert into <for-distribution log>
  select getdate(), 'i' , * from inserted
end
```

The *<ownership predicate>* is generated based on the *ownership type* of each table, and will usually reference the name of the current site. We assume that this is available in the SQL variable `@site`.

We illustrate next the different ownership types with simplified examples from an application for planning extensions and changes to the telephone network.[3] Telenor is divided into regions, each responsible for its own part of the network. Apart from data describing equipment within their region, planners need only limited, read-only access to other relevant data, usually from neighbor regions. This partitioning of responsibility is directly reflected in the *ownership* and *distribution* rules.

We use the following tables in the examples. Key columns are underlined.

```
TABEXPORT(TABNAME, OWNER, RECIP )
TABOWNER(TABNAME, OWNER)
ORGUNIT(ORGID, OWNER, NAME)
SWITCH(SID, ORGID, TYPE, NAME)
ROUTING(SID, RNO, FROMNO, TONO, DATE)
TRUNK(SI1, SI2, TNO, ORGID, TYPE, NAME)
TRUNKPROG(TI1, TI2, TNO, YEAR, PNO, PROGN)
```

For a graphical presentation of tables and ownership dependencies, see figure 2. Dark columns indicate primary key for each table. Circled attributes are used for ownership fragmentation.

4.1 Entire ownership

A table may be owned completely by some site, i.e. the ownership fragment is the entire table contents. The owner site can be determined at compilation time, or installation time. In our example, the table TABOWNER is owned by a central site called

[1] This general form can in many cases be (and is actually) optimized.

[2] MODIFIED is a table alias name for the deleted and inserted virtual tables of Sybase triggers.

[3] The simplification consists of removing attributes that are not relevant to the examples, and replacing composite keys by single attributes.

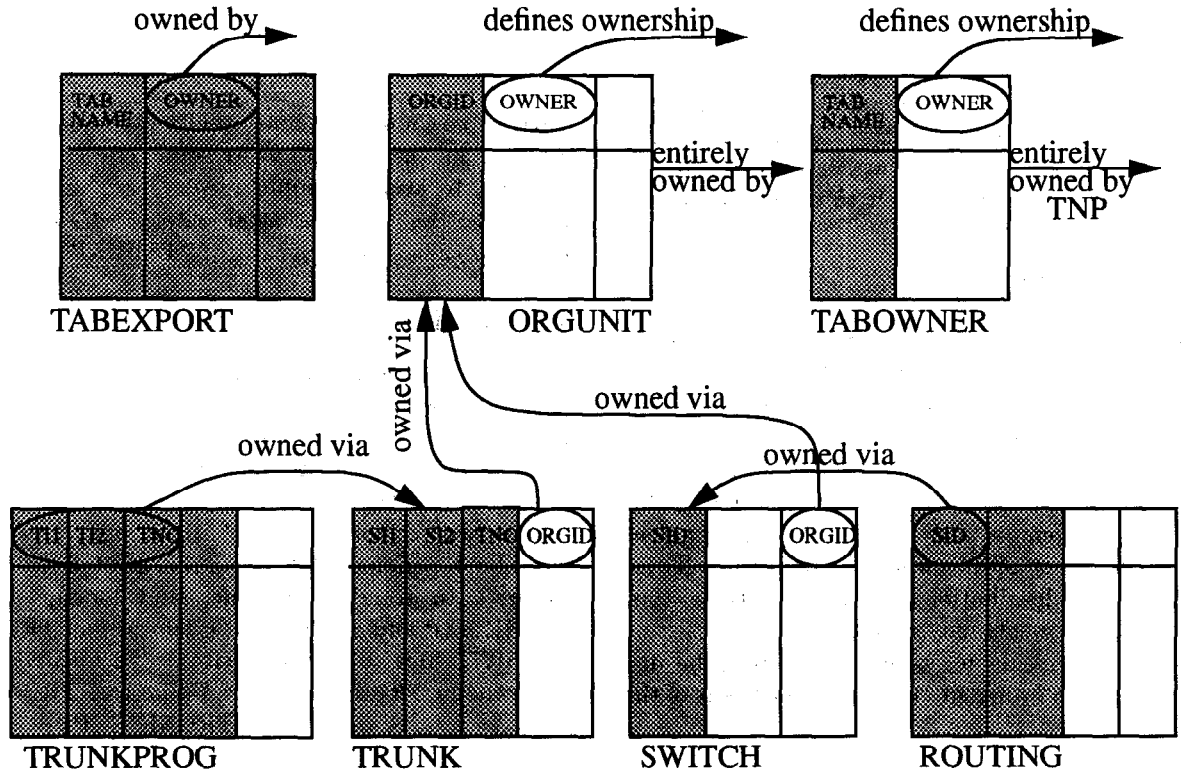

Figure 2: Sample tables with ownership.

'TNP'. This give an *ownership predicate* of the form: $\mathcal{O}_{TABOWNER} \equiv$ 'TNP' = @*site*

TABOWNER is a dictionary table which maps table names to owner sites. ORGUNIT is a table which maps organizational units to sites, and is owned entirely by a single site: $\mathcal{O}_{ORGUNIT} \equiv \pi_{OWNER}(\sigma_{TABNAME='ORGUNIT'}\text{TABOWNER}) = \{@site\}$

4.2 Direct ownership

TABEXPORT is a dictionary table which is used for describing distribution of other tables from an owner site to recipient sites. The TABEXPORT table is owned directly based on the value of the OWNER attribute. This value based ownership assures that each owner site can determine whether tables should be replicated to other sites. This gives an *ownership predicate* of the form: $\mathcal{O}_{TABEXPORT} \equiv \pi_{OWNER}(\text{MODIFIED}) = \{@site\}$

4.3 Indirect ownership

Indirect/via lookup table. Responsibility for switches and trunk lines are allocated to organizational units within Telenor. This responsibility is directly reflected in the tables SWITCH(..., ORGID) and TRUNK(..., ORGID). The ORGID attribute is a foreign key on the ORGUNIT table. Ownership for SWITCH and TRUNK is based on this foreign key, and ORGUNIT.OWNER gives the owner site. Both tables have an *ownership predicate* of the form: $\mathcal{O}_{SWITCH} \equiv \mathcal{O}_{TRUNK} \equiv \pi_{OWNER}(\text{ORGUNIT} \bowtie \text{MODIFIED}) = \{@site\}$

The ownership lookup table (ORGUNIT) is owned by a central site (see above) in order to ensure global ownership consistency.

Indirect/transitive. The SWITCH and TRUNK tables are "top level" tables in two separate hierarchies of foreign key dependencies, with lots of tables containing different data about the various parts of switches and trunks. TRUNKPROG, which describes trunk prognosis, is an example of this. Ownership is based on a foreign key, and the fact that ownership via TRUNK is transitive. The *ownership predicate* becomes: $\mathcal{O}_{TRUNKPROG} \equiv \pi_{OWNER}(\text{ORGUNIT} \bowtie \text{TRUNK} \bowtie \text{MODIFIED}) = \{@site\}$

Indirect ownership can be transitive in several levels, limited only by the SQL compiler of the target system. In practice we have never needed more than two levels. The joins generated for transitive ownership predicates are compiled and stored in the database as triggers, so they make an excellent match for the technique of *join indices* and its generalizations [Val87, KM90], if this is supported by the target system.

4.4 Arbitrary ownership

In addition to the ownership types described above, we allow arbitrary SQL expressions as ownership predicates. This works much like an "assembler" directive in e.g. a C compiler: any executable code can be inserted, but the correctness is left to the programmer.

5 Distribution types

As for ownership, we identify common distribution cases and call them *distribution types*. Based on these types we generate distribution predicates that are stored in a dictionary table for use by the data distribution application.

Distribution predicates are used to select data for distribution to remote sites. A distribution fragment is always a subset of an ownership fragment. There may be several distribution rules for each table, thus distribution fragments may overlap.

An SQL expression for selecting tuples to be distributed from an owner site to a recipient site is of the form:[4]

```
select * from <for-distribution log>
where TIMESTAMP >
      ( <time of previous distribution> )
  and ( <ownership predicate> )
  and (    (<distribution predicate 1> )
        or (<distribution predicate 2> )
        or ... )
order by TIMESTAMP
```

This assumes that the triggers have stored all tuples which are to be replicated in *for-distribution log* tables. There is one log table for each replicated table. The query will be evaluated at the owner site, and each tuple returned is converted to a delete or an insert operation at the recipient site.

The <*ownership predicate*> can be removed because we know that the log contains only tuples which are owned locally. Each <*distribution predicate*> in the following sections is evaluated against the *for-distribution log* tables, except at system "boot-strap", where we read base tables restricted by *ownership predicate* AND *distribution predicate(s)*.

5.1 Entire distribution

Entire to all. Tables ORGUNIT and TABOWNER are distributed to all sites. The entire ownership fragment is distributed. This type of distribution is used for tables which are maintained centrally, and should be replicated to all sites. This gives a trivial *distribution predicate* of TRUE.

Entire to some. Some tables should be distributed in their entirety to some sites. This is used e.g. to keep a complete overview of SWITCHes and TRUNKs at a central cite. $\mathcal{D}_{TRUNK} \equiv \sigma_{TABNAME='TRUNK',RECIP=@recipient}$TABEXPORT

5.2 Indirect distribution

Indirect/via lookup table. Owner sites correspond to regions within Telenor. Application users in each region will need to import information about TRUNK lines crossing region borders, and SWITCHes at the end of these lines, i.e. regions will export parts of their data to other (neighbor) regions. This is implemented with an "indirect to some sites" rule, and lookup tables TRUNK_EXP and SWITCH_EXP which map TRUNKs and SWITCHes to recipient sites. The *distribution predicate* for TRUNK becomes: $\mathcal{D}_{TRUNK} \equiv$ TRUNK $\bowtie$ ($\sigma_{OWNER=@site,RECIP=@recipient}$TRUNK_EXP)

Referring to the previous section, we see that there are two (independent) distribution predicates for TRUNK. The complete distribution predicate will be the union of the two.

Ownership of the distribution lookup tables (TRUNK_EXP and SWITCH_EXP) should reflect business policy: *Entire* owhership will centralize the control of replication, *direct* ownerhip based on a SENDER attribute will assure that each owner site can determine the fragments to be replicated to other sites.

Indirect/transitive. As with ownership rules, distribution rules may also be transitive. If a site exports data about one of its trunks to a given recipient site, the corresponding trunk prognosis in TRUNKPROG should follow. This gives a *distribution predicate*: $\mathcal{D}_{TRUNKPROG} \equiv$ TRUNKPROG $\bowtie$ TRUNK $\bowtie$ ($\sigma_{OWNER=@site,RECIP=@recipient}$TRUNK_EXP)

5.3 Arbitrary distribution

As with ownership, we also allow arbitrary SQL expressions for distribution predicates. These are "safe" in the sense that they are always restricted with the *ownership predicate*, so that we can guarantee that only locally owned data is ever exported.

5.4 Incremental replication and indirect distribution

The data imported to a site may be regarded as a union of read-only snapshots from other sites. These snapshots are maintained incrementally, even in the

[4] Actually the query is transformed and optimized. Replacing the disjunction with unions, and precomputing inner queries using SQL variables gave a speedup of two orders of magnitude for query evaluation at the sender site.

case of indirect distribution, where these snapshots are *joins*. Incremental replication is implemented by cascading any modifications of distribution lookup tables to the *for-distribution logs* of the dependent tables. Given the distribution predicates for TRUNK and TRUNKPROG (see above), we extend the trigger of TRUNK_EXP to log necessary dependent tuples in TRUNK and TRUNKPROG. Similarly the trigger of TRUNK is extended to log dependent TRUNKPROG tuples. This allows us to always evaluate *distribution predicates* against the *for-distribution log tables*, rather than the base tables.

6 Current implementation

Ownership and distribution rules are defined using our SQL extensions, as shown in the examples of sections 4 and 5, and the appendix. Our compiler takes the declarative specifications and generates a collection of database triggers to prevent modifications to imported data, and log changes. These triggers are installed in each of the local databases. We have so far worked on Sybase databases, but have plans for an Oracle version. The most significant difference between Oracle and Sybase is that Oracle has *row triggers*, rather than the set-oriented triggers of Sybase.

Performance measurements show an average increase in response times of about 50% for database modification requests which do *not* violate the ownership tests.[5] This should be acceptable for many classes of applications, assuming that only a small fraction of the execution time is spent on actual database modifications. In our example applications most of the time is spent querying the database, and by the users browsing data on their screens.

We have a separate application at each site to perform the incremental update of replicas. It is executed periodically and also in response to explicit "import data" or "export data" requests. This application uses the distribution predicates generated by the compiler and the *for-distribution* logs maintained by the triggers. Replica update consists of several transactions with two-phase commit, each transaction updating *one* table on *one* recipient site with the modifications from *one* sender site. We do not update all tables in a single transaction because our experiments showed excessive lock contention with conflicting local user transactions. Alternatives for data replication are limited by the trigger model/interface of Sybase: Triggers are evaluated as the transaction progresses, and have no access to either a transaction id or a transaction commit time. Therefore, in our current system modifications done by a transaction are not propagated atomically to other sites, and *fragmentwise serializability* is not guaranteed (see section 3). Serializability of local transactions is enforced by the local Sybase DBMS.

The performance of the distribution application is determined by three factors: Evaluating distribution predicates at the sender site, data transfer, and insertion at the recipient site. We insert data at the recipient site using precompiled stored procedures, so data transfer and insertion is faster than local insertion using SQL INSERT statements. We have optimized the distribution predicates (se footnote on page 7) and obtained response times of a couple of seconds for tables with indirect distribution, and *for-distribution* logs containg 1000 tuples (on a SPARCstation 1).

Analyzing *locality* of a system (see section 3) reduces to the question: Does each site have enough information to determine the tuples it owns and the tuples it needs to replicate? Our ownership and distribution types facilitate answering these questions by giving explicit means to trace data dependencies. For example, we know that if a table is owned entirely or directly (see section 4), then ownership of tuples can be decided locally. When the ownership is via other tables, the data that affects ownership of any tuple is well defined. Consistency is achieved through design, and not guaranteed automatically by the use of the special types we have defined. But such types simplify the design. A situation to watch for is when local decisions depend on data owned by other sites. The designer must ensure that such data is actually distributed to the sites where it is needed.

Our special ownership and distribution types also help in the analysis of distribution, to make sure the system converges in a finite time to an appropriate state. Again, the key property is that data dependencies are clearly identified, and that consistency is achieved through design, thus obviating the need for sophisticated rule confluence analysis [vdVS93].

DIBAS has been successfully applied to two projects within Telenor, both are in the "acceptance test" phase at the time of writing. The owner and distribution types described above have been developed as a response to the user and application needs of these two projects. A language and a corresponding compiler was implemented in order to experiment with different SQL expressions for the ownership and distribution predicates. Writing a "distribution schema" in this language required close cooperation between the designers of the centralized database schema, and the DIBAS team. In addition to the compiler, we have developed a set of tools for "boot-strapping" the decentralized database. Normal operation of the decentralized database requires only ordinary Sybase DBA tasks at each individual site. We have how-

[5] Single *insert*, *delete*, or *update* operations of tables with direct or indirect ownership. The increase in response time for transitively owned tables (TRUNKPROG in our examples) is about 100%.

ever implemented some extra tools for monitoring the *for-distribution* logs. We also provide utilities for distinguishing between owned and imported tuples at a database site:[6]

```
create view table_mine as
select table.* from table
where <ownership predicate>
create view table_not_mine as
select table.* from table
where not <ownership predicate>
```

Our fragmentation and de-centralization assumes that most applications will only update locally owned data. It is however possible to modify imported data. One possibility is to update the data at the owner site, and perform a re-distribution. In order to update remote data, the application can either execute a remote procedure call (RPC) from the local server,[7] or it can open connections to (any number of) remote servers directly.[8]

6.1 Alternative asynchronous replication schemes

Asynchronous replication seems to be a trend of commercial DBMS vendors. Sybase [Syb93], Ingres [ASK93], Oracle [Ora93] and others have recently introduced replication along the lines used by DIBAS — but they do not provide the facilities and degree of control offered by our approach. Sybase reads the transaction log, and propagates replicated data according to "subscriptions", which may project/select data from individual tables. Subscriptions are manually coded. Ingres logs changes to base tables in "shadow and archive tables" which are similar to our *for-distribution* log tables. Changes are logged by rules that are generated based on a menu of table/column selections. Ingres allows rules to project/select data for replication. In DIBAS the replication is modelled in a high-level language, which is translated into SQL queries. These queries can replicate entire tables or selections of tables, and also use lookup tables (transitively) to allow replication of individual tuples based on foreign-key dependencies.

To our knowledge, none of the commercial asynchronous replication systems have any notion of ownership fragmentation. Preventing conflicting updates is usually much cheaper and more effective than detection and resolution after the fact, especially in a peer-to-peer replication environment. Commercial DBMS's typically provide discretionary acces on tables and columns. Compared to our *ownership types*, this means that only tables with "entire" ownership can be safely distributed. Triggers and rules to enforce read-only access to imported data would have to be coded manually. For "direct ownership" this is trivial, but it would be quite error prone and tedious for the "indirect" case, especially for transitive ownership.

[6] These are read-only views, since Sybase does not allow updates through views of this complexity.

[7] Sybase RPC's execute as separate transactions.

[8] Sybase supports two-phase commit transaction control for applications which open several database connections.

7 Conclusions

In this paper we presented the problem of database de-centralization, which is driven by increasingly common restructuring needs of companies. A key issue is the flexible, quick reassignment of responsibilities over data and cost of equipment. DBMS vendors currently provide some components of a solution, but there is no comprehensive and user-friendly approach to de-centralization. The DIBAS project is delivering concepts, techniques, and tools to address this problem.

We proposed an architecture based on commercially available technology, which requires no modifications to existing applications or underlying DBMS. We described ownership and distribution types that facilitate the design of the distributed system. Such types are declaratively specified using SQL extensions, and converted automatically into appropriate triggers. This allows a compact, readable and maintainable notation for fragmentation and replication. The DIBAS de-centralization approach has been tested in practice at a large company that is going through a re-structuring phase.

References

[ASK93] ASK Group, Alameda, CA. *ASK OpenINGRES Replicator User's Guide*, December 1993.

[AWH92] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data, San Diego, California*, pages 59–68, 1992.

[CP85] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, New York, 1985.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane*, pages 566–577, 1990.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance.

In *Proceedings of the Seventeenth International Conference on Very Large Databases, Barcelona*, pages 577–589, 1991.

[CW93] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proceedings of the Nineteenth International Conference on Very Large Databases, Dublin*, pages 108–119, 1993.

[GMK88] H. Garcia-Molina and B. Kogan. Achieving high availability in distributed databases. *IEEE Transactions on Software Engineering*, 14(7):886–896, July 1988.

[HHW89] A. A. Heddaya, M. Hsu, and W.E. Weihl. Two phase gossip: Managing distributed event histories. *Information Sciences*, 49(1,2,3):35–57, Oct./Nov./Dec. 1989. Special issue on databases.

[HW93] E. N. Hanson and J. Widom. An overview of production rules in database systems. *The Knowledge Engineering Review*, 8(2):121–143, June 1993.

[KM90] A. Kemper and G. Moerkotte. Access support in object bases. In *Proceedings of ACM-SIGMOD 1990 International Conference on Management of Data, Atlantic City, New Jersey*, pages 364–374, 1990.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[MD89] D. R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data, Portland, Oregon*, pages 215–224, 1989.

[Ora93] Oracle Corporation. *Oracle7 Symmetric Replication*, September 1993. White paper.

[OV91] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching, and views in database systems. In *Proceedings of ACM-SIGMOD 1990 International Conference on Management of Data, Atlantic City, New Jersey*, pages 281–290, 1990.

[SL90] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[Sto92] M. Stonebraker. The integration of rule systems and database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):415–423, October 1992.

[Syb90] Sybase, Inc., Emeryville, CA. *Sybase Commands Reference, Release 4.2*, May 1990.

[Syb93] Sybase, Inc., Emeryville, CA. *Sybase Replication Server, release 10.0*, August 1993.

[Val87] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[vdVS93] Leonie van der Voort and Arno Siebes. Enforcing confluence of rule execution. In *Proceedings of the 1st International Workshop on Rules in Database Systems, Edinburgh*, pages 194–207, 1993.

[WCL91] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Databases, Barcelona*, pages 275–285, 1991.

[ZH90] Y. Zhou and M. Hsu. A theory for rule triggering systems. In *Advances of Database Technology —EDBT'90, LNCS 416*, pages 207–421, Berlin, March 1990. Springer-Verlag.

A Fragmentation language, and generated SQL code

In this appendix we show ownership fragmentation rules, distribution rules, and generated SQL code for the examples. Ownership and distribution rules are shown on the left hand side, generated SQL code on the right hand side. For brevity we show only the generated <*ownership predicate*> and <*distribution predicate*> not the entire trigger or selection expression.

Entire ownership

A table which is owned entirely by a single site is owned via the dictionary table TABOWNER. Alternatively, we can specify the name of the owner site directly.

```
ownership for TABOWNER
owned entirely by TNP
```

```
'TNP' = @site
```

```
ownership for ORGUNIT
owned entirely
```

```
exists (
  select * from TABOWNER
  where TABNAME = 'ORGUNIT'
    and OWNER   = @site )
```

Direct ownership

We specify the name of the attribute which determines ownership.

```
ownership for TABEXPORT
owned directly
given by OWNER
```

```
MODIFIED.OWNER = @site
```

Indirect/via lookup table

We specify the attributes of the foreign key, and the name of the lookup table. For the lookup table we must specify how it determines ownership for other tables. In this case, ownership of *dependent* tables is different from the *defining* table.

```
indirect ownership via ORGUNIT
given by OWNER
```

```
exists (
  select * from ORGUNIT
  where ORGUNIT.OWNER = @site
    and MODIFIED.ORGID  = ORGUNIT.ORGID )
```

```
ownership for TRUNK, SWITCH
owned indirectly
given by ORGID via ORGUNIT
```

Indirect/transitive

We specify the attributes of the foreign key, and the name of the lookup table. For the lookup table we specify transitive ownership, i.e. ownership of *dependent* tables is the same as the *defining* table.

```
indirect ownership via TRUNK
```

```
exists (
  select * from TRUNK
  where  exists (
    select * from ORGUNIT
    where ORGUNIT.OWNER = @site
      and TRUNK.ORGID = ORGUNIT.ORGID )
    and MODIFIED.TI1 = TRUNK.SI1
    and MODIFIED.TI2 = TRUNK.SI2
    and MODIFIED.TNO = TRUNK.TNO )
```

```
ownership for TRUNKPROG
owned indirectly
given by TI1, TI2, TNO
via TRUNK
```

Entire distribution

We specify that the table should be replicated to all sites, or to a set of selected sites (found in the dictionary table TABEXPORT).

```
distribution for ORGUNIT
distribute to all
```

```
1 = 1
```

```
distribution for TRUNK
distribute to some
```

```
0 < ( select count(*) from TABEXPORT
      where TABEXPORT.TABNAME = 'TRUNK'
        and TABEXPORT.RECIP   = @recipient
        and TABEXPORT.OWNER   = @site     )
```

Indirect/via lookup table

We specify the attributes of the foreign key, and the name of the lookup table. For the lookup table we must specify how it determines distribution for other tables. In this case, distribution of *dependent* tables is different from the *defining* table. We have two independent distribution rules for TRUNK, which generates a disjunction.

```
indirect distribution via TRUNK_EXP
given by RECIP
```

```
0 < ( select count(*) from TABEXPORT
      where TABEXPORT.TABNAME = 'TRUNK'
        and TABEXPORT.RECIP   = @recipient
        and TABEXPORT.OWNER   = @site     )
or
exists (
  select * from TRUNK_EXP
  where  TRUNK_EXP.RECIP = @recipient
    and LOG_TRUNK.SI1 = TRUNK_EXP.SI1
    and LOG_TRUNK.SI2 = TRUNK_EXP.SI2
    and LOG_TRUNK.TNO = TRUNK_EXP.TNO )
```

```
distribution for TRUNK
distribute indirectly
given by SI1, SI2, TNO
via TRUNK_EXP
```

Indirect/transitive

We specify the attributes of the foreign key, and the name of the lookup table. For the lookup table we specify transitive distribution, i.e. distribution of *dependent* tables is the same as the *defining* table.

```
indirect distribution via TRUNK
```

```
0 < ( select count(*) from TABEXPORT
      where TABEXPORT.TABNAME = 'TRUNK'
        and TABEXPORT.RECIP = @recipient
        and TABEXPORT.OWNER = @site     )
or
exists (
  select * from TRUNK_EXP
  where  TRUNK_EXP.RECIP = @recipient
    and TRUNK.SI1 = TRUNK_EXP.SI1
    and TRUNK.SI2 = TRUNK_EXP.SI2
    and TRUNK.TNO = TRUNK_EXP.TNO )
  and LOG_TRUNKPROG.TI1 = TRUNK.SI1
  and LOG_TRUNKPROG.TI2 = TRUNK.SI2
  and LOG_TRUNKPROG.TNO = TRUNK.TNO )
```

```
distribution for TRUNKPROG
distribute indirectly
given by TI1, TI2, TNO
via TRUNK
```