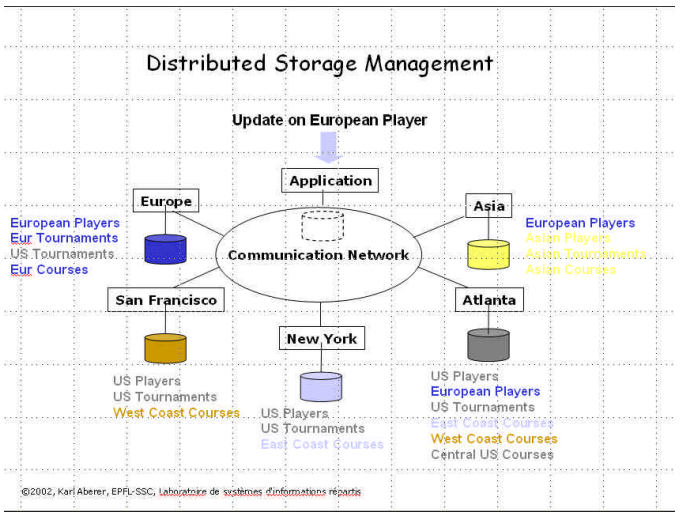
Distributed Data Management
Part 1 - Schema Fragmentation

Today's Question

1. Distribution of Relational Databases
2. Horizontal Fragmentation of Relational Tables
3. Vertical Fragmentation of Relational Tables

Problem 3. Scalability for Large Databases

- Locate data on available storage medium in an efficient manner
  - Blocks, files, network nodes, ...
- Provide efficient access to data for specific addressing methods (Indexing)
  - attributes, predicates, paths, ...
  - Data access structure (tree, hash table etc.)
- Example: B+-Tree
  - tree nodes match block size of storage systems
  - tree is balanced
  - all operations (search, update) logarithmic



1. Relational Databases

DEPARTMENTS

DNo	DName	Budget	Location
P4	Sales	500000	Geneva
P2	Marketing	300000	Paris
P3	Development	250000	Munich
P1	Development	150000	Bangalore
P5	Marketing	120000	Geneva
P6	Development	90000	Paris
P7	Research	80000	Paris

EMPLOYEES

ENo	EName	Title	DNo
E1	Smith	Manager	P1
E2	Lee	Director	P1
E3	Miller	Assistant	P2
E4	Davis	Assistant	P3
E5	Jones	Manager	P3

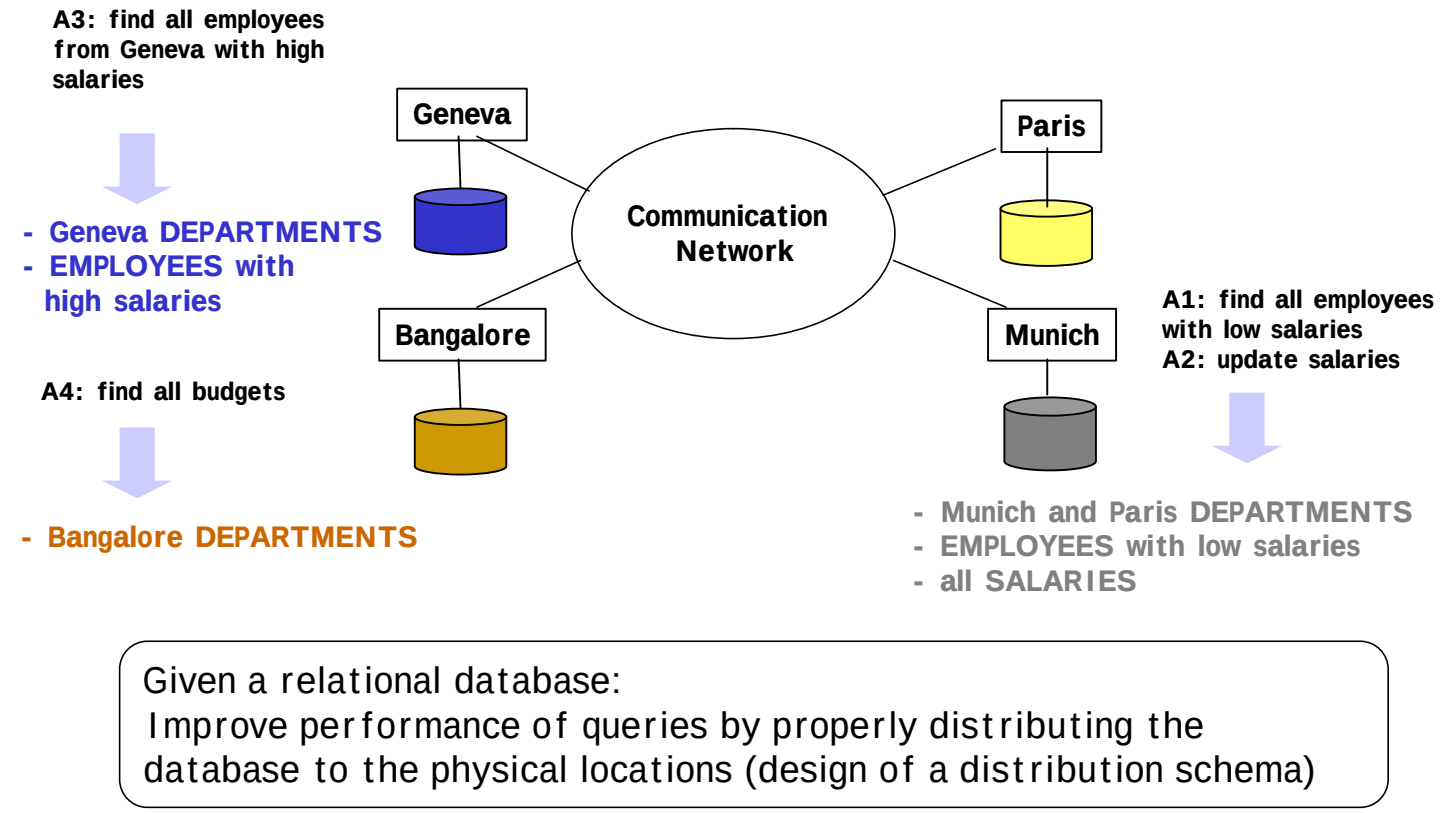
SALARIES

Skill	Salary
Director	200000
Manager	100000
Assistant	50000

In this lecture we introduce some basic concepts on distributing relational databases. These concepts exhibit some important principles related to the problem of distributing data management in general. In particular, these principles apply beyond the relational data model.

In the following we assume that we are familiar with the basic notions of the relational data model, including the notions of relation, attribute, query, primary and foreign keys, relational algebra operators, and relational calculus (in particular the notion of predicates and basic logical operators).

Distributed Relational Databases



Given a relational database:
Improve performance of queries by properly distributing the database to the physical locations (design of a distribution schema)

Having a (relational) database that is shared by distributed applications in a network opens immediately a possibility to optimize the access to the database, when taking into account which applications at which sites require which data with which frequency. An obviously useful idea is to move the data to the place where it is needed most, in order to reduce the cost of accessing data over the network. The communication cost involved in accessing data over the network is high as compared to local access to data.

The example illustrates the situation, where the relational database from the previous slide is distributed to the sites where the database is accessed (applications are indicated by A1-A4). The distribution schema, i.e. the description which parts of the database are distributed to which site, and the applications (queries) are given informally. A possible reasoning for distributing the data as shown could be as follows: since A3 needs all information about Geneva departments and high salary employees we put the related data in that site. In Bangalore only the DEPARTMENT table is accessed, but parts of it are allocated to other sites as they are used there, therefore only the locally relevant data is kept. Paris has no applications, so no data is put there. Munich has all other data, in particular, for example, the salary table, which is also used in Geneva, but more frequently in Munich, as it is updated there.

In the following we will introduce methods of how to make the specification of such a distribution schema precise and provide algorithms that support the process of developing such a distribution schema.

What Do You Think ?

- Problems to solve when distributing a relational database

Assumptions on Distribution Design

- Distributed relational database system is available
  - allow to distribute relational data over physically distributed sites
  - takes care of transparent processing of database accesses
- Top-down design
  - no pre-existing constraints on how to organize the database
- Access patterns are known and static
  - no need to adapt to changes in access patterns (otherwise redesign)
- Replication is not considered
  - reasonable assumption if updates are frequent

The problem of distributing a relational database is a very general one: we will make a number of assumptions in order to be able to focus on specific questions. We will not concern ourselves with the issue of developing a distributed database system architecture. This requires to solve a number of important problems, such as communication support, management of the data distribution schema, and processing of distributed queries. We assume that if we can specify of how the data is to be distributed all other issues are taken care of. Thus we focus on the problem of distribution schema design.

We also assume that there exist no a-priori constraints on how we distribute the database, be it of technical or organizational nature. We are free to decide which data goes where.

The access patterns are assumed to be static, or changing so slowly that we can afford to perform a re-design whenever needed. Thus we can design our distribution schema off-line.

Finally we do not take advantage of replication, which is a reasonable assumption in update-intensive environments. Methods involving replication can pursue similar approaches as we will describe, but considering it introduces an additional design dimension which for the purpose of clarity we will ignore.

Degree of Fragmentation

- Complete relations are too coarse, single attribute values are too fine
  - Determine proper parts (fragments) of relations
  - Idea: use a SQL query against a relation to specify fragments
- Example

SELECT Dno, DName FROM DEPARTMENT WHERE Budget > 200000

vertical fragment

horizontal fragment

DNo	DName	Budget	Location
P4	Sales	500000	Geneva
P2	Marketing	300000	Paris
P3	Development	250000	Munich
P1	Development	150000	Bangalore
P5	Marketing	120000	Geneva
P6	Development	90000	Paris
P7	Research	80000	Paris

s

A first important question is: to which degree should fragmentation occur, i.e. which parts of a relation can be distributed independently. We will call these parts in the following "fragments". Restricting the distribution to complete relations appears to be too limited in general, in particular when considering tables containing information relevant for different sites. On the other extreme deciding on the distribution for each single attribute value or tuple seems to be a too complex task when considering the distribution of very large tables. A flexible way to create fragments that can be distributed to the sites is to use queries which can select subsets of a relational table, as shown in the example. These fragments can be essentially of two different kinds:

1. **horizontal fragments** of a table are defined through selection (i.e. what is specified in the WHERE clause of a SQL query). These are subsets of tuples of a relation.
2. **vertical fragments** of a table are defined through projection (i.e. what is specified in the SELECT clause of a SQL query). These are subtables consisting of a subset of the attribute columns.

Correct Fragmentation

- Completeness
  - decomposition of a relation R into fragments R1,…,Rn is complete if every attribute value found in one of the relations is also found in one of the fragments
- Reconstruction
  - if a relation is decomposed into fragments R1,…,Rn then it should also be possible to reconstruct the relation R from its fragments (e.g. by applying appropriate relational operators such as join, union etc.)
- Disjointness
  - if a relation is decomposed into fragments R1,…,Rn then every attribute value should be contained only in one of the fragments
- Attention: Reconstruction and (full) disjointness cannot be achieved at the same time (more later)

When decomposing a relational table into fragments a number of minimal requirements have to be satisfied in order to avoid the loss of information. First, we have to make sure that every data value of the original table is found in one fragment, otherwise we loose this data value. This property is called **completeness**. Second, we must be able to **reconstruct** the original table from the fragments. This is a problem very similar to the one encountered when normalizing relational database schemas by decomposition of tables. Also there it can occur that by improper decomposition we can no more reconstruct the original table. Finally, the fragments should be **disjoint** (in order to avoid update dependencies) as far as possible. We will see later that the last two conditions of reconstruction and disjointness can not be completely satisfied at the same time in general.

Summary

- Why should a relational database be fragmented ?
- At which phase of the database lifecycle is fragmentation performed ?
- What are the alternative approaches to fragment relations ?
- Under which conditions is a fragmentation considered correct ?
- In which environments would replication be an appropriate alternative to fragmentation ?

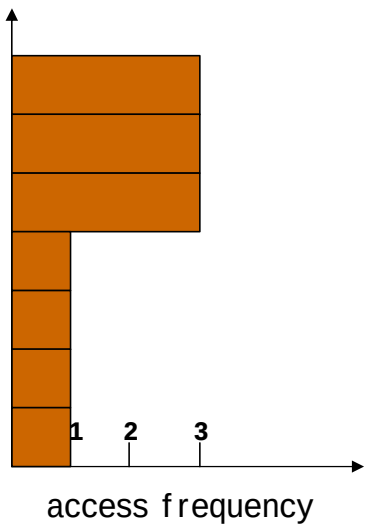
2. Primary Horizontal Fragmentation

- *Horizontal* Fragmentation of a *single* relation
- Example
  - Application A1 running at Geneva:
    "update department budgets > 200000 three times a month, others monthly"

fragment F1
budget > 200000

fragment F2
budget <= 200000

DNo	DName	Budget	Location
P4	Sales	500000	Geneva
P2	Marketing	300000	Paris
P3	Development	250000	Munich
P1	Development	150000	Bangalore
P5	Marketing	120000	Geneva
P6	Development	90000	Paris
P7	Research	80000	Paris



We begin by fragmenting a single relational table horizontally. Since the fragmentation of a table depends on the usage of the table, we have first to be able to describe of how the table is accessed. In other words, we need a model for the table access. One possible model would be to give for every single tuple the frequency of access of a specific application, as illustrated in the histogram on the right. Since we do however not consider fragmentation of a table into single tuples, this description is at a too fine granularity. Also one can see that for many tuples the access frequency will be the same (as a consequence of the structure of the application executing an SQL query)

Thus we rather model the access only for those parts of relations that potentially qualify for fragmentation. Thus the model we are interested in has to describe two things: first what are possible (horizontal) fragments about which we want to say something, and second what we want to say about the access. The answer to the first question is a consequence of our idea of using SQL to describe fragments: the horizontal fragment will be described by some form of predicate (or logical expression) that consists of conditions on the attributes of the table. As for the second question, we restrict ourselves to specifying that for a given fragment the tuples are all accessed with the same frequency. This corresponds exactly to what we see in the example. We have two fragments F1 and F2 described by a predicate and all tuples in each of the fragments are accessed with uniform frequency.

Modeling Access Characteristics

- Describe (potential) horizontal fragments
  - select subsets of the relation using predicates
- Describe the access to horizontal fragments
  - all tuples in a fragment are accessed with the same frequency
- Obtain the necessary information
  - provided by developer
  - analysis of previous accesses

The necessary information on access frequencies either can be provided by a developer, who knows the application and can derive from that the necessary (approximate) specification, or is obtained from analysis of database access logs. The second approach is technically more challenging, and typically will require statistical analysis or data mining tools (we will introduce basic data mining techniques at the end of this lecture)

Determining Access Frequencies

- What do we need to know about horizontal fragments ?
  - access frequency af(Ai, Fj): given a tuple in Fj, how often is it accessed by application Ai per time unit

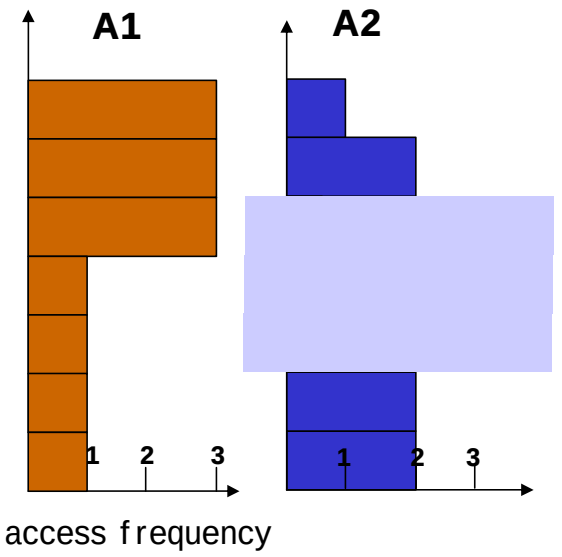
- Examples:
  - update of some tuple in Fj by Ai occurs t times per time unit:
    af(Ai, Fj) = t/size(Fj)
  - query by Ai accesses all tuples in Fj and query occurs t times per time unit:
    af(Ai, Fj) = t
  - query by Ai accesses 10% of all tuples in Fj and query occurs t times per time unit:
    af(Ai, Fj) = t/10

The access frequencies are measured in terms of average number of accesses to a tuple of the fragment within a time unit. Thus each access to each tuple is counted as a single access for an application. This information can be derived from the application in different ways, as is illustrated in the examples.

Example Multiple Applications

- A second application A2 running in Paris:
  - "request the Bangalore dept budget on average three times a month"
  - "request some Geneva dept budget twice a month"
  - "request some Paris dept budget 6 times a month"
  - "request Munich dept budget every second month"

DNo	DName	Budget	Location
P4	Sales	500000	Geneva
P2	Marketing	300000	Paris
P3	Development	250000	Munich
P1	Development	150000	Bangalore
P5	Marketing	120000	Geneva
P6	Development	90000	Paris
P7	Research	80000	Paris



If we can describe the access to a relation by one application we can do the same also for other applications as shown in the example. For a single relation we know what are the potential horizontal fragments, those that are identified by the access model as having same access frequency. With multiple applications we see that there exist different possible combinations of access frequencies for different tuples since different applications fragment the relations differently. Each different combination potentially might lead to a different decision on where to locate the tuple.

Example Access Frequencies to Fragments

- Each fragment can be described as conjunction of predicates, e.g.

  F1: Location = "Paris" $\wedge$ Budget > 200000

- There exist the following different combinations of access frequencies <af1, af2> for applications A1 and A2

<af1, af2>	Location = "Paris"	Location = "Geneva"	Location = "Munich"	Location = "Bangalore"
Budget > 200000	<3, 2>	<3, 1>	<3, 0.5>	n/a
Budget <= 200000	<1, 2>	<1, 1>	n/a	

F1

Important Observation: if all tuples in a set of tuples are accessed with the same frequency by all applications, then whichever method we use to optimize access to tuples, these tuples will be assigned to the same site

Therefore it makes no sense to make a further distinction among them, i.e. fragments in which all tuples are accessed with equal probability are the smallest we have to consider

What we can do is to enumerate all possible combinations of access frequencies as shown in this example. We take each possible combination of horizontal fragments from the two applications. If we form the conjunction of the predicates describing the fragments in each of the applications (which can be done by using the logical AND connector), then we obtain fragments of the relational tables for which the access frequency is the same for all tuples for both applications. We find these access frequencies thus in the entries of the table capturing all possible combinations of predicates.

An important observation relates now to the fact that we have not to further fragment the table than it is done by combining all possible fragments of all applications, since whichever method we use to distributed the tuples to different sites, it will not be able to distinguish them (through the access frequency) and thus they will be moved to the same site.

Describing Horizontal Fragments

- **(Simple) predicates P**: testing the value of a single attribute
  - Examples: P = {Location = "Paris", Budget > 200000}
- **Minterm predicates M(P):** Combining *all* simple predicates taken from P using "and" and "not" ($\wedge$ and $\neg$)
  - Example:
    If P = {Location = "Paris", Budget > 200000, DName = "Sales"}
    then Location = "Paris" $\wedge$ $\neg$Budget > 200000 $\wedge$ DName = "Sales"
    is 1 of 8 possible elements in M(P)

Formally: Given a relation R[A1, …, An], then a simple predicate p is
p: Ai op Value
where op ä {=, <, <=, >, >=, not =} and Value ä Di, Di domain of Ai

Formally: Given R[A1, …, An] and simple predicates P = {p1, …, pm}, then the set M(P) of minterm predicates consists of all predicates of the form

$$\wedge_{pi \in P} pi^*$$

where pi* is either pi or ¬pi.

As we have seen we need conjunctions of predicates in order to describe fragments in the general case. We make now the description of horizontal fragments more precise: Given a relation we can assume that there exists a set of atomic predicates that can be used to describe horizontal fragments, these are called simple predicates. From those we can compose complex predicates by using conjunctions and negations. More precisely we consider all possible compositions of all simple predicates using conjunction and negation. This set we call **minterm predicates** and it constitutes the set of all predicates that we consider for describing horizontal fragments.

One might wonder why disjunctions (OR) are not considered. In fact they would be of no use as they would allow to only define fragments that are the union of some fragments one obtains from minterm predicates. In other words with minterm predicates we obtain the finest partitiong of the relational table that can be obtained by using a given set of simple predicates, and this is sufficient to describe the access frequencies for all tuples.

Horizontal Fragments

- A *horizontal fragment* Fi of a relation R consists of all tuples that satisfy a minterm predicate mi

- Example:
  m1 : Location = "Paris" ∧ ¬ Budget > 200000 ∧ DName = "Research"
  m2 : ¬ Location = "Geneva" ∧ Budget > 200000

DNo	DName	Budget	Location	
P4	Sales	500000	Geneva	
P2	Marketing	300000	Paris	F2
P3	Development	250000	Munich	
P1	Development	150000	Bangalore	
P5	Marketing	120000	Geneva	
P6	Development	90000	Paris	
P7	Research	80000	Paris	F1

All possible horizontal fragments are those subsets of a relation that can be selected by using a minterm predicate over a given set of simple predicates.

Complete and Minimal Fragmentation

- How many simple predicates do we need ?
  - e.g. is P={Budget > 200000, Budget <= 200000} a good set ?

- *At least* as many such that the access frequency within a horizontal fragment is uniform for all tuples for all applications (otherwise we could not model the access)
  -> *complete* set of simple predicates

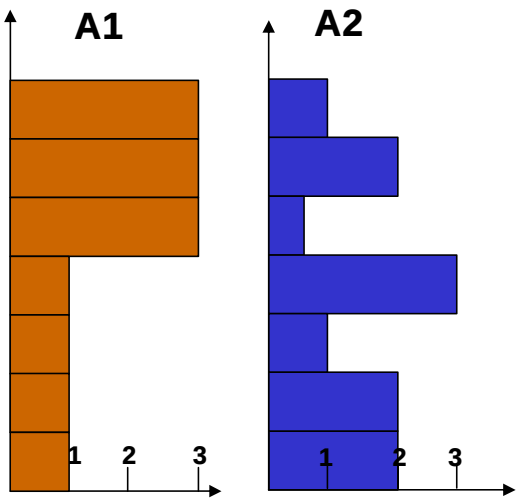
- *but no more*
  -> *minimal* set of simple predicates

The situation is now as follows. Different applications will use (propose) different simple predicates in order to describe the access to a relation. They will need as many simple predicates as necessary, to obtain fragments for which the access frequency for the specific application is uniform. As we have seen in order to describe the combined access frequencies of multiple applications to the relations we have to combine those simple predicates into complex predicates. Thus possible fragments are constructed from minterm fragments over the set of simple predicates that is the union of the set of all simple predicates used by the different applications. This set allows to construct any possible intersection of fragments originating from different applications through minterm predicates. However, a set of simple predicates obtained in this manner can contain simple predicates that are not useful, such that we would consider too many minterm predicates which lead to no additional fragments.

Example

- P1 = {Location = "Paris", Budget > 200000 } not complete
- P2 = {Location = "Paris", Location = "Munich", Location = "Geneva", Budget > 200000 } complete, minimal ?
- P3 = {Location = "Paris", Location = "Munich", Location = "Geneva", Budget > 200000, Budget <= 200000 } , complete but not minimal

DNo	DName	Budget	Location
P4	Sales	500000	Geneva
P2	Marketing	300000	Paris
P3	Development	250000	Munich
P1	Development	150000	Bangalore
P5	Marketing	120000	Geneva
P6	Development	90000	Paris
P7	Research	80000	Paris



We illustrate the difference between complete and minimal set of predicates in this example. P2 is not complete since it does not allows e.g. to distinguish Geneva from Munich, which have different access frequencies for A1 and A2. P3 is obviously not minimal. The question is whether P2 is complete and minimal.

Example Minimal Fragmentation

F1 : Location="Paris" ∧ ¬ Location="Geneva" ∧ Budget > 200000
F2 : Location="Paris" ∧ ¬ Location="Geneva" ∧ ¬ Budget > 200000
F3 : ¬ Location="Paris"∧ Location="Geneva" ∧ Budget > 200000
F4 : ¬ Location="Paris"∧ Location="Geneva" ∧ ¬ Budget > 200000
F5 : ¬ Location="Paris"∧ ¬ Location="Geneva" ∧ Budget > 200000
F6 : ¬ Location="Paris"∧ ¬ Location="Geneva" ∧ ¬ Budget > 200000

<AF1, AF2>	Location = "Paris"	Location = "Geneva"	Location = "Munich"	Location = "Bangalore"
Budget > 200000	<3, 2> (F1)	<3, 1> (F3)	<3, 0.5> (F5)	n/a
Budget <= 200000	<1, 2> (F2)	<1, 1> (F4)	n/a	<1, 3> (F6)

- P2' = {Location = "Paris", Location = "Geneva", Budget > 200000 } is

Here we see that actually the predicate Location = "Munich" is not needed. The observation is that for the given database this predicate is only useful to distinguish F5 and F6, but this can already be done with another predicate, namely Budget>20000. Therefore P2 is in fact complete and it is minimal since we cannot eliminate any further simple predicate from P2 without loosing completeness.

It is very important to understand that this fact depends on the actual state of the database, i.e., the content of the relation. As soon as for example a tuple would enter the database, which contains a department in Munich with budget less than 200000, the predicate Location = "Munich" will be needed to describe the new fragment (provided it is accessed differently than current fragment F6).

Determining a Minimal Fragmentation

- Given fragments generated by a set M(P):
  We say a predicate p is *relevant* to M(P) if there exists at least one element of m ™ M(P), such that when creating the fragments corresponding to m1 = m ∧ p and m2 = m ∧ ¬ p there exists at least one application that accesses the two fragments F1 and F2 generated by m1 and m2 differently

Algorithm ***MinFrag***

Start from a complete set of predicates P

Find an initial p ™ P such that p relevant to M({})

set P' = {p}, P = P \ {p}

Repeat until P empty

- find a p ™ P such that p relevant to M(P')
- set P' = P' Y {p}, P = P \ {p}
- if there exists a p ™ P' that is not relevant to M(P' \ {p}) then set P'=P' \ {p}

The algorithm MinFrag determines a minimal set of simple predicates from a given set and for a given database. It proceeds by iteratively adding predicates from the given complete set of predicates. While doing that it observes two things: first, it adds only predicates that are relevant, with respect to the currently selected set of predicates. This is expressed by the concept of RELEVANCE. Second, in each step it checks whether one of the already included predicates has become non-relevant through the addition of the new predicate. In fact it might be the case that one predicate p1 is "more relevant" than another p2 included earlier, i.e. we can eliminate p2 without loosing interesting fragments but not vice versa.

Example MinFrag Algorithm

- P3 = {Location = "Paris", Location = "Munich", Location = "Geneva", Budget > 200000, Budget <= 200000} is a complete set of predicates

Step 1: add Location = "Munich"	(ok)
Step 2: add Budget > 200000	(ok)
Step 3: add Budget <= 200000	(no, is dropped)
Step 4: add Location = "Paris"	(ok)
Step 5: add Location = "Geneva"	(ok, but now Location = "Munich" is dropped)

We illustrate of how the MinFrag algorithm would work for our example. The dropping of the predicate in step 3 is for obvious reasons. In step 5 the predicate Location="Munich" is dropped, since as we have seen earlier it is not required to distinguish all possible fragments. Note, that in case Location="Munich" would have only been considered in the last step (rather in the first), it would never have been included into the set P'. Thus the execution of the algorithm depends on the order of processing of predicates from the initial complete set of predicates.

Eliminating Empty Fragments

- Not all minterm predicates constructed from a complete and minimal set of predicates generate useful fragments
- Example:
  {Location = "Paris", Location = "Geneva", Budget > 200000 } is minimal
- All minterm predicates

F1 : Location="Paris" ∧ ¬ Location="Geneva" ∧ Budget > 200000
F2 : Location="Paris" ∧ ¬ Location="Geneva" ∧ ¬ Budget > 200000
F3 : ¬ Location="Paris"∧ Location="Geneva" ∧ Budget > 200000
F4 : ¬ Location="Paris"∧ Location="Geneva" ∧ ¬ Budget > 200000
F5 : ¬ Location="Paris"∧ ¬ Location="Geneva" ∧ Budget > 200000
F6 : ¬ Location="Paris"∧ ¬ Location="Geneva" ∧ ¬ Budget > 200000
F7 : Location="Paris" ∧ Location="Geneva" ∧ Budget > 200000
F8 : Location="Paris" ∧ Location="Geneva" ∧ ¬ Budget > 200000

Finally, after executing MinFrag, it still is possible that certain minterm fragments are to be excluded for logical reasons. It is very well possible as illustrated in this example that we need a certain minimal set of simple predicates in order to properly describe all horizontal fragments of the relation, but that we can construct from this set minterm predicates that produce empty fragments, as shown in the example. The typical example is where multiple equality conditions on the same predicate are included. Then the conjunction of two such predicates in their positive form (unnegated) always leads to a contradictory predicate, resp. an empty fragment.

Summary Primary Horizontal Fragmentation

- Properties
  - Relation is completely decomposed
  - We can reconstruct the original relations from fragments by union
  - The fragments are disjoint (definition of minterm predicates)

- Application provides information on
  - what are fragments of single applications
  - what are the access frequencies to the fragments

- Algorithm MinFrag
  - derives from a complete set of predicates a minimal set of predicates needed to decompose the relation completely
  - without producing unnecessary fragments

Derived Horizontal Fragmentation

DEPARTMENTS

DNo	DName	Budget	Location
P4	Sales	500000	Geneva
P2	Marketing	300000	Paris
P3	Development	250000	Munich
P1	Development	150000	Bangalore
P5	Marketing	120000	Geneva
P6	Development	90000	Paris
P7	Research	80000	Paris

← horizontal fragment

EMPLOYEES

ENo	EName	Title	DNo
E1	Smith	Manager	P1
E2	Lee	Director	P1
E3	Miller	Assistant	P2
E4	Davis	Assistant	P3
E5	Jones	Manager	P3

derived horizontal fragment →

foreign key

Since the process of fragmenting a single relation horizontally is a considerable effort, the question is whether such a fragmentation cannot be exploited further. In fact, there exists a good reason to do so when considering of how typically relational database schemas are constructed. In general, one finds many foreign key relationships, where one relation refers to another relation by using it's primary key as reference. Since these relationships carry a specific meaning it is very likely, that this foreign key relationship will also be used during accesses to the database, i.e. by executing join operations over the two relations. This means that the corresponding tuples in the two relations will be jointly accessed. Thus it is of advantage to keep them at the same site in order to reduce communication cost.

As a consequence it is possible and of advantage to "propagate" a horizontal fragmentation that has been obtained for one relation to other relations that are related via a foreign key relationship and to keep later the corresponding fragments at the same site. We call fragments obtained in this way **derived horizontal fragments**.

Semi-Join

Formally: Given relations R[A1,…,Ar] and S[B1,…,Bs] where Aj is primary key of R and Bi is a foreign key of S referring to Aj.

Given a horizontal fragmentation of R into R1,…,Rk then this induces the derived horizontal fragmentation

Sn = S ▷ Rn, n=1,..,k

Semi-Join: S ▷ R = $p_{B1,…,Bs}$(S▷◁R)

π	**projection**
▷◁	**natural join**

Formally the derivation of horizontal fragments can be introduced using the so-called **semi-join operator**. The semi-join operator is a relational algebra operator, that takes the join of two relations, but then projects the result to one relation. When computing the semi-join of a horizontal fragment with another relation one obtains the corresponding derived horizontal fragments of the second relation.

Multiple Derived Horizontal Fragmentations

- Distribute the primary and derived fragment to the same site
  - tuples related through foreign key relationship will be frequently processed together (relational joins)
- If multiple foreign key relationships exist multiple derived horizontal fragmentations are possible
  - choose the one where the foreign key relationships is used most frequently

DNo	DName	Budget	Location
P4	Sales	500000	Geneva
P2	Marketing	300000	Paris
P3	Development	250000	Munich
P1	Development	150000	Bangalore
P5	Marketing	120000	Geneva
P6	Development	90000	Paris
P7	Research	80000	Paris

Skill	Salary
Director	200000
Manager	100000
Assistant	50000

ENo	EName	Title	DNo
E1	Smith	Manager	P1
E2	Lee	Director	P1
E3	Miller	Assistant	P2
E4	Davis	Assistant	P3
E5	Jones	Manager	P3

In general, different DHFs can be obtained if the same relation is related to multiple relations through a foreign key relationship. In that case a decision has to be taken, since a fragmentation according to different primary fragmentation would make no sense: it would not be possible to keep the tuples in the derived fragments together with the corresponding primary fragments if they are moved to different sites. Therefore the DHF is chosen, which is induced by the relation that is expected to be used most frequently together with the relation for which the DHF is generated.

Summary

- How are horizontal fragments specified ?
- When are two fragments considered to be accessed in the same way ?
- What is the difference between simple and minterm predicates ?
- How is relevance of simple predicates determined ?
- Is the set or predicates selected in the MinFrag algorithm monotonically growing ?
- Why are minterm predicates eliminated after executing the MinFrag algorithm ?