Implementation of Databases Notes

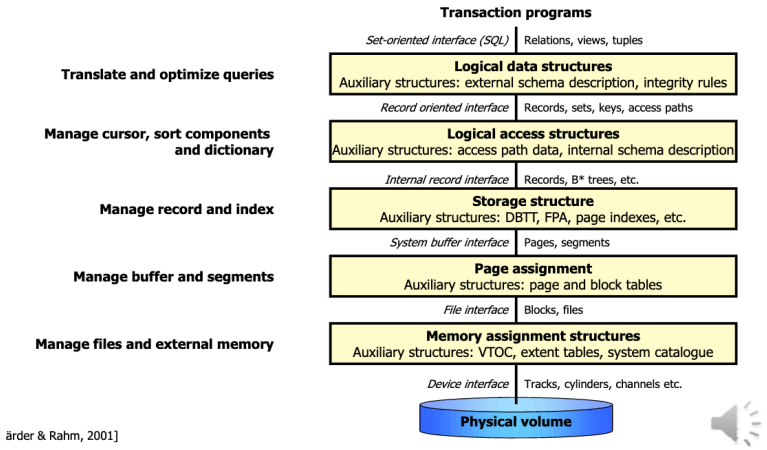
Table of Contents

1. Architecture of Database Systems

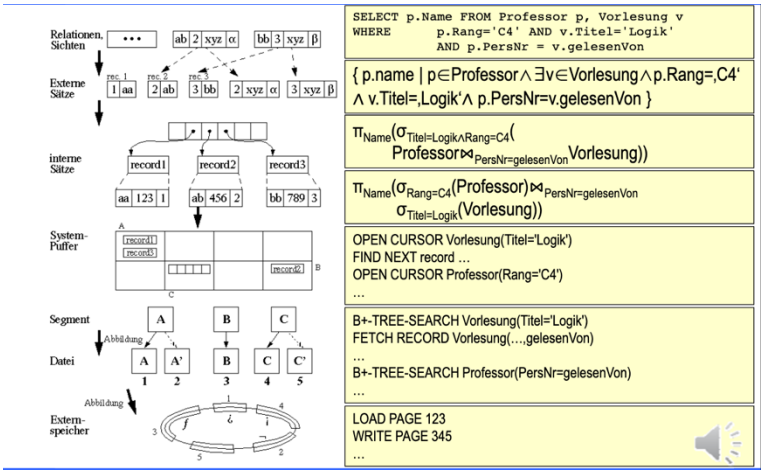
1.1 Goals and Tasks of DBMS

- **Data independence** is the main goal of DBMS. It means that data is managed independent of applications. It refers to the immunity of user applications to changes made in the definition and organisation of data. It makes data available for different applications. There are two types of data independences.
  - **Physical data independence**: logical schema is independent of physical structure, i.e., relational schema is independent of changes on indexes, clustering, etc.
  - **Logical data independence:** external schema is independent of logical schema, i.e., relational views are defined as derived relations on top of logical schema (the relational schema with the base relations); logical schema might change while external schema does not need to be changed.
- **Five layers**
  - **Logical Data Structure** is mainly for translating and optimising queries. The addressing units between this layer and Transaction Programs are views, tuples and tables. The auxiliary structure is external schema description. And the addressing units to the lower level are external records, sets, keys and access paths. The interface between transaction programs or users and Logical Data Structure is Set-Oriented Interface (SQL).
  - **Logical Access Structure** is mainly for managing cursors, sorting components and managing dictionaries. The auxiliary structures are access path data and internal schema description. The addressing units to the lower layer are internal records, B* trees and so on. The interface between Logical Data Structure and Logical Access Structure is Record Oriented Interface for offering logical access path to individual records.
  - **Storage Structure** is responsible for managing records and indexes. The auxiliary structures are DBTT, FPA, page indexes and so on. The addressing units to the lower layer include page and segments. The interface between Logical Access Structure and Storage Structure is Internal Record Interface, in which records are stored in B* tree.
  - **Page Assignment** is for managing buffers and segments. The auxiliary structures are page and block tables. The addressing units to the lower layer include blocks and files. The interface between Storage Structure and Page Assignment is System Buffer Interface.
  - **Memory Assignment Structure** is responsible for managing files and external memories. The auxiliary structures are VTOC, extent tables and system catalogue. The addressing units to Physical Volume are tracks, cylinders, channels and so on. The interface between Page Assignment and Memory Assignment Structure is File Interface. And the interface between Memory Assignment Structure and Physical Volume is Device Interface.

1.2 Basic Architecture of DBMS



ärder & Rahm, 2001]



1.3 Distributed Database Systems

A **distributed database (DDB)** is a collection of multiple, logically interrelated databases distributed over a computer network. The software system that permits the management of the distributed database and makes the distribution transparent to the users is called **distributed database management system (D–DBMS)**. **Transparency** is the separation of higher level semantics of a system from the lower level implementation issues. A fundamental issue in DDBs is to provide **data independence** in the distributed environment. Many times scalability is achieved at the expense of consistency.

a distributed database system (DDBS) is the conjuction of both. **DDBS = DDB + D–DBMS**

Some of the promises of D-DBMS are:

- A **transparent** management of distributed, fragmented, and replicated datal
- Improved **reliability**/availability through distributed transactions.
- Improved **performance**.
- Easier and more **economical** system expansion → Scalability.

An **Information Resource Dictionary** is a shareable repository for a definition of the information resources relevant to all or part of an enterprise.

2. Advanced Transaction Management

2.1 Definitions

1. A **transaction(TX)** is a DB program, which only consists of read and write operations to a database. These operations are denoted as read(x)or write(x), where xis a DB object.

2. Let $D = \{x, y, z \ldots\}$ be a database. Then a **transaction** $t(TX)$ is a finite series of operations in the form $r(x)$ („read x“) or w(x) („write x“) denoted as $t = p_1, \ldots, p_n$ with $n < \infty, p_i \in \{r(x), w(x)\}$ for $1 \leq i \leq n$ and $x \in D$. Indices are used to distinguish various (concurrent) transactions.

3. Let $T = \{t_1, \ldots, t_n\}$ be a (finite) set of transactions. Thus

   - $shuffle(T)$ is the Shuffle Product of T. (the sum of all ways of interlacing them, e.g. $ab \cdot xy = abxy + axby + xaby + axyb + xayb + xyab$)
   - a **complete schedule** s for T is a serie $s' \in shuffle(T)$ with the additional pseudo actions c_i (commit) and a_i (abort) for each $t_i \in T$ according to the following rules:
     1. $(\forall i, 1 \leq i \leq n) c_i \in s \Leftrightarrow a_i \notin s$
     2. $(\forall i, 1 \leq i \leq n) c_i$ or a_i are in s, whereever, but after the last action of t_i
   - $shuffle_{ac}(T)$ is the set of all complete schedules
   - a **schedule** is a prefix of a complete schedule
   - a complete schedule is **serial,** if for a permutation ρ from {1,...,n} it holds that all the transactions run one after the other without any interference with one another (no e.g. dirty read) they are terrible from a performance point of view because we have to wait until a transaction finishes executing to execute the next one:

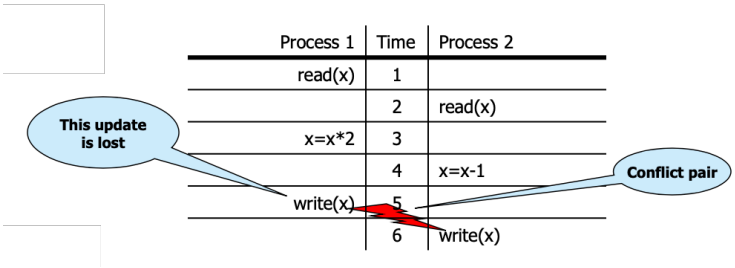
$$s = t_{\rho(1)} \ldots t_{\rho(n)}$$

4. Notation for Schedule s

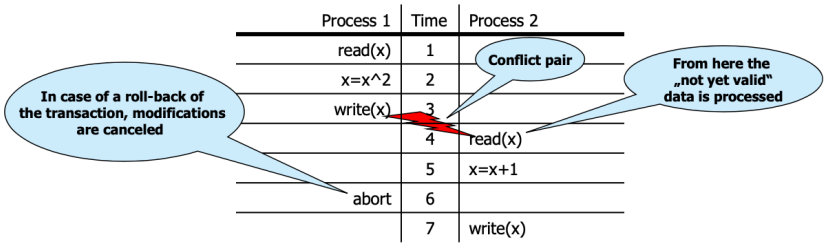
   - trans(s) = $\{t_i | s$ contains actions of $t_i\}$
   - commit(s) = $\{t_i \in$ trans(s) $| c_i \in s\}$
   - abort(s)= $\{t_i \in$ trans(s) $| a_i \in s\}$
   - active(s)= trans(s) – (commit(s) ∪ abort(s))
   - op(s) = set of all the actions occurring in s

2.2 Synchronization Problems

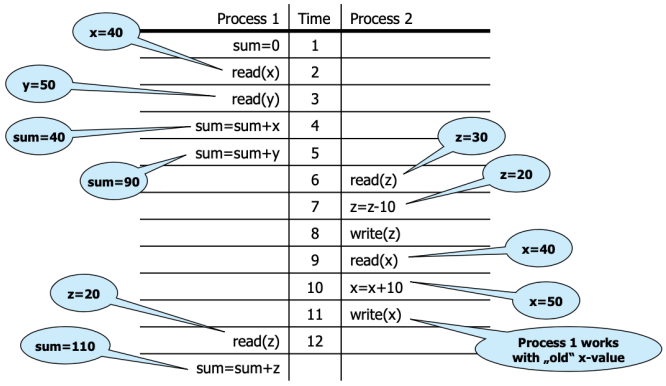
- Lost update



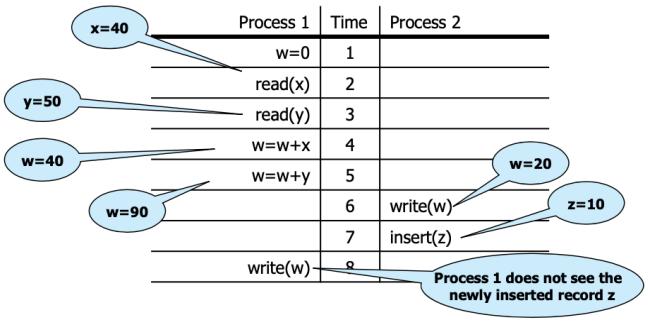
- Dirty read



- Non-repeatable read/Inconsistent read



- Phantom Problem



ACID Principle

Every transaction must be processed in the way that the ACID properties are preserved.

- **Atomicity**: In an execution of a transaction, either all operations are carried out, or none are.
- **Consistency**: Preservation of all integrity constraints of the DB, i.e. a transaction starts with a consistent DB state, and after the execution of the transaction the DB state is consistent as well.
- **Isolation**: Isolated execution of a transaction, i.e. „as if executed solely"
- **Durability**: Once a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes

2.3 Serializability Theory

Definitions

Let s and s' be schedules. s and s' are called **final-state equivalent,** denoted as $s \approx_f s'$, if $op(s) = op(s')$ and all DB objects have at the end identical values in s and s', according to the abstract semantics.

A schedule s is called **final-state serializable** if there exists a serial schedule s' which is final-state equivalent to s. FSR is the class of all final-state serializable schedules.

Example 1:
$s = r1(x)r2(y)w1(y)r3(z)w3(z)r2(x)w2(z)w1(x)$ and
$s' = r3(z)w3(z)r2(y)r2(x)w2(z)r1(x)w1(y)w1(x)$.

In s: $x = f_{1x}(x_0)\ y = f_{1y}(x_0)\ z = f_{2z}(x_0, y_0)$
In s': $x = f_{1x}(x_0)\ y = f_{1y}(x_0)\ z = f_{2z}(x_0, y_0)$
$\Rightarrow s \approx_f s' \Rightarrow s \in FSR$

Example 2:
$s = r_1(x)r_2(y)w_1(y)w_2(y)c_1c_2$
$s' = r_1(x)w_1(y)r_2(y)w_2(y)c_2c_1$

In s: $y = f_{2y}(y_0)$
In s': $y = f_{2y}(f_{1y}(x_0))$
$\Rightarrow s \approx_f s' \Rightarrow s \notin FSR$

2.4 Conflict Serializability Classes

Let s be a schedule, $t, t' \in$ trans(s) and $t \neq t'$:

- Two data operations $p \in t$ and $q \in t'$ in s are in **conflict,** if they operate on the same object and at least one of them is a write operation.
- $C(s) = \{(p, q) | p, q$ in s are in conflict and p is before q in $s\}$ are the **conflict relations** of s.

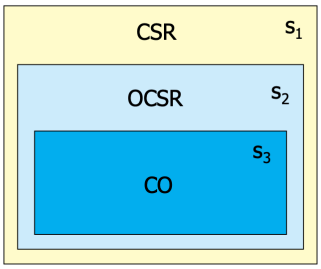
Example
Let $s = w_1(x)r_2(x)w_2(y)r_1(y)w_1(y)w_3(x)w_3(y)c_1a_2$.

Then: $C(s) = \{(w_1(x), r_2(x)), (w_1(x), w_3(x)), (r_2(x), w_3(x)), (w_2(y), r_1(y)), (w_2(y), w_1(y)), (w_2(y), w_3(y)), (r_1(y), w_3(y)), (w_1(y), w_3(y))\}$.

$\Rightarrow conf(s) = (w_1(x), w_3(x)), (r_1(y), w_3(y)), (w_1(y), w_3(y))$. Conflict relations with an operation of transaction 2 have been removed.

$conf(s)$ denotes the conflict relations of a schedule s, which are cleaned up by aborted transactions.

Three Serializability classes will be presented: CSR, $OCSR$ and CO.

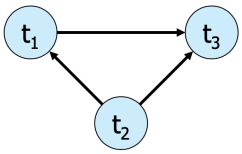


with
- $s_1 = w_1(x)\ r_2(x)\ c_2\ w_3(y)\ c_3\ w_1(y)\ c_1 \in CSR - OCSR$
- $s_2 = w_3(y)\ c_3\ w_1(x)\ r_2(x)\ c_2\ w_1(y)\ c_1 \in OCSR - CO$
- $s_3 = w_3(y)\ c_3\ w_1(x)\ r_2(x)\ w_1(y)\ c_1\ c_2 \in CO$

CSR

- Let s and s' be two schedules. s and s' are called **conflict equivalent**, denoted as $s \approx_c s'$, if:
  - $op(s) = op(s')$ and
  - $conf(s) = conf(s')$.
- A complete schedule s is called **conflict serializable**, if a serial schedule s' exists with $s \approx_c s'$.
- The **conflict graph**

$s = r_1(x)\ r_2(x)\ w_1(x)\ r_3(x)\ w_3(x)\ w_2(y)\ c_3\ c_2\ w_1(y)\ c_1$



Theorem 2.2:
$s \in CSR \Leftrightarrow G(s)$ is acyclic.
(Because the transitions can be ordered t_2, t_1, t_3 in the example)
Membership in CSR can be tested in polynominal time

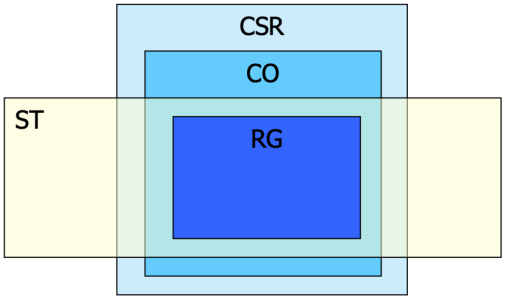
OCSR

A complete schedule s is called **order-preserving conflict serializable**, there exists a serial schedule s' with $s \approx_c s'$ and the following holds for all $t, t' \in trans(s)$:
If t occurs completely before t' in s, then the same holds in s'.

CO

A schedule s is called **commit order-preserving conflict serializable** (or owns the property of **commit order preservation**), if the following holds:
For all $t_i, t_j \in commit(s), i \neq j$, with $(p, q) \in conf(s)$ for $p \in t_i, q \in t_j$,
then : c_i is before c_j in s.

2.5 Recovery Theory



RC

A schedule s is called **recoverable**, if the following holds:
$(\forall t_i, t_j \in trans(s), i \neq j)$ t_i reads from t_j in $s \wedge c_i \in s \Rightarrow c_j <_s c_i$
(If transaction 2 reads from transaction 1, then transaction 1 commits before transaction 2)
„Every transaction will not be released, until all other transactions from which it has read, are released."

Example:
Let $s1 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)c_2w_1(z)c_1$
It holds: t_2 reads y from t_1 and $c_2 \in s$, but $c_1 \prec c_2$. Consequently $s_1 \notin RC$

Let $s_2 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)w_1(z)c_1c_2$
It holds: $s_2 \in RC$, because the commit operation of t_2 is after the one of t_1 , but the abort of t_1 leads to the abort of t_2, this may give rise to cascading aborts.

ACA

A schedule s **avoids cascading aborts**, if it holds:
$(\forall t_i, t_j \in trans(s), i \neq j)\ t_i$ reads x from t_j in $s \Rightarrow c_j <_s r_i(x)$
„A transaction is only allowed to read values from already successfully completed transactions."

Example:
$s2 = w1(x)w1(y)r2(u)w2(x)r2(y)w2(y)w1(z)c1c2 \notin ACA$
$s3 = w1(x)w1(y)r2(u)w2(x)w1(z)c1r2(y)w2(y)c2 \in ACA$
Further problem: The values, which are restored after an abort, may be different from the Before Images of the write operations of the aborting transactions.

ST

A schedule s is called **strict**, if the following holds:
$(\forall t_i \in trans(s))(\forall p_i(x) \in op(t_i), p \in r, w)$
$w_j(x) <_s p_i(x), i \neq j \Rightarrow a_j <_s p_i(x) \vee c_j <_s p_i(x)$
„A schedule is strict, if an object is not read or overwritten, until the transction, which has written it at last, is terminated." (Same as before but now also with the write operation)

Example:
$s_3 = w_1(x)w_1(y)r_2(u)w_2(x)w_1(z)c_1r_2(y)w_2(y)c_2 \notin ST$
$s_4 = w_1(x)w_1(y)r_2(u)w_1(z)c_1w_2(x)r_2(y)w_2(y)c_2 \in ST$

RG

A schedule s is called **rigorous**, if it is strict and satisfies the following condition:
$(\forall t_i, t_j \in trans(s))r_j(x) <_s w_i(x), i \neq j \Rightarrow a_j <_s w_i(x) \vee c_j <_s w_i(x)$
„A schedule is rigorous, if it is strict and no object x is overwritten, until all transactions, which have read x at last, are terminated."

- A schedule from ST avoids Write-Read- as well as Write-Write conflicts between non-released transactions
- A schedule from RG avoids additionally Read-Write conflicts between those kinds of transactions.

2.6 Scheduling Algorithms

Techniques with which a DBMS can generate correct schedules for transactions to be processed; these are called scheduling protocols, or in short **scheduler**. A scheduler gets as input a sequence of operations (r,w,a,c) and it must produce a correct output schedule from them.

Locking Scheduler

The scheduler can apply locks for the synchronization of accesses on data objects that are used together. There are two types of locks for an object x: - Read lock: rl(x) *read lock*, ru(x) *read unlock* - Write lock: wl(x) *read lock*, wu(x) *read unlock*

Rules for the application of locks
For each t_i, which is contained completely in a schedule s, the following should be valid:

1. If t_i contains a $r_i(x)[w_i(x)]$, thus $rl_i(x)[wl_i(x)]$ stands anywhere before it in s and $ru_i(x)[wu_i(x)]$ stands anywhere after it.
2. For each x processed by t_i there are exactly one $rl_i(x)$ resp. $wl_i(x)$ in s
3. No ru_i/wu_i is redundant

Examples:
$s_1 = rl_1(x)r_1(x)ru_1(x)wl_2(x)w_2(x)wl_2(y)w_2(y)wu_2(x)wu_2(y)c_2wl_1(y)w_1(y)wu_1(y)c_1$
$s_2 = rl_1(x)r_1(x)wl_1(y)w_1(y)ru_1(x)wu_1(y)c_1wl_2(x)w_2(x)wl_2(y)w_2(y)wu_2(x)wu_2(y)c_2$

A scheduler **works according to a locking protocol**, if for every output s and every $t_i \in trans(s)$ it holds: - t_i satisfies the rules 1. to 3 for the application of locks. - If x is locked by t_i and $t_j, t_i, t_j \in trans(s), i \neq j$, then these locks are compatible

Two Phase Locking (2PL)

A locking protocol is **two phase**, if for every generated schedule s and every transaction $t_i \in trans(s)$ it holds:
After the first ou_i action there is no further ql_i action $(o, q \in \{r, w\})$. Such a scheduler is called a **2PL scheduler**.
"In the first phase of a transaction locks will only be set, in the second phase locks will only be removed."

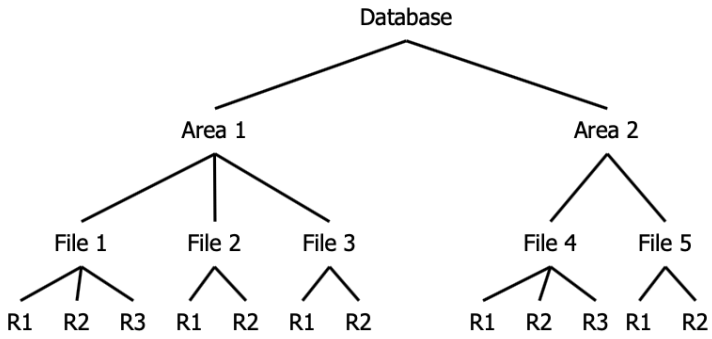
Examples:
$s_1 = rl_1(x)r_1(x)ru_1(x)wl_2(x)w_2(x)wl_2(y)w_2(y)wu_2(x)wu_2(y)c_2wl_1(y)w_1(y)wu_1(y)c_1$
s_1 is not 2PL. $s_2 =$
$rl_1(x)r_1(x)wl_1(y)w_1(y)ru_1(x)wu_1(y)c_1wl_2(x)w_2(x)wl_2(y)w_2(y)wu_2(x)wu_2(y)c_2$
s_2 is 2PL

Theorem 2.2
$\varepsilon(2PL) \subseteq CSR$

Variants of 2PL - Conservative 2PL : All locks are available since BOT - Strict 2PL (S2PL): Hold all write locks till EOT - Strong 2PL (SS2PL): Hold all locks till EOT

Theorem 2.3
$\varepsilon(S2PL) \subseteq CSR \cap ST$

Multi-Granularity Locking (MGL)



- Each transaction can choose the suitable granularity by itself. (in the example below: record file, table space, area, database) (You can choose to lock the entire File 1 or Area 2 for example)(If something below is locked, you can't lock above, that's where intention locks come in handy)
- The scheduler must then prevent transactions from setting conflicting locks in overlapping granularities.

If the database is tree-structured, two provisions are helpful: - Distinction between explicit and implicit locks (higher-level locks implicitly lock also lower level objects) - Propagation of locks in tree upwards as **intention locks** (irl, iwl, $riwl$)

Each transaction t_i is locked/unlocked as follows: 1. If x is not the root of the database, t_i must own a ir- or iw-lock on the parent node of x, in order to be able to set $rl_i(x)$ or $irl_i(x)$. 2. If x is not the root of the database, t_i must own a iw-lock on the parent node of x, in order to be able to set $wl_i(x)$ or $iwl_i(x)$. 3. To read (write) x, t_i must own a r-lock or w-lock on x. 4. t_i cannot remove an intentional lock on x, as long as t_i has still a lock on a child of x.

Summary: *Locks are set top-down and removed bottom-up.*
We can prove that, for every transaction, which keeps the 2-Phase rules, $\varepsilon(MGL) \subseteq CSR$ is valid.

Index Locking

Assumption so far: - DB is a fixed collection of independent objects - Even Strict 2PL might not guarantee serializability if objects are added during a transaction.

Example: (Phantom Problem, assume page-level locking is used)

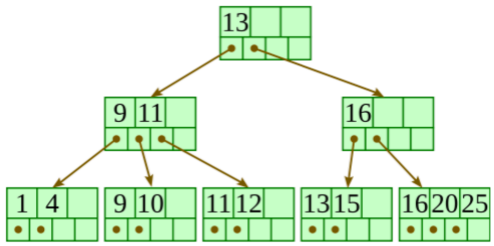
1. T1 locks all pages containing person records with sex=male, and finds oldest person (e.g. age=71)
2. T2 inserts a new male person with age=96
3. This record is inserted on a different page than the pages locked by T1
4. T2 deletes oldest female person with age=80

5. This record is also located on a page which is not locked by T1
6. T2 commits
7. T1 now locks all pages containing female person records and finds oldest (e.g. age=75)

⇒ There is no consistent DB state where T1 is correct!

- T1 implicitly assumes that it has locked the set of all male person records
  - This is true only if no records are added while T1 is executed. Thus, some mechanism to enforce this assumption is needed.
- The example shows that conflict serializability is guaranteed only if the set of objects is fixed.
- Possible Solutions
  - No Index: T1 has to lock all pages and the file/table to prevent new records/pages being added – very inefficient!
  - Index on sex field:
    - T1 needs to lock the index page with data entries for sex=male
    - If there are no such records yet, T1 must lock the index page where such a data entry would be created.

B+ Trees and the Simple Locking Algorithm



A B+-tree of type $(k, k*)$ is a multi-path tree with the following properties:

- Every node has one more references than it has keys.
- All leaves are at the same distance from the root.
- For every non-leaf node N with k being the number of keys in N: all keys in the first child's subtree are less than N's first key; and all keys in the ith child's subtree $(2 \leq i \leq k)$ are between the $(i-1)$th key of n and the ith key of n.
- The root has at least two children.
- Every non-leaf, non-root node has at least $floor(d/2)$ children.
- Each leaf contains at least $floor(d/2)$ keys.
- Every key from the table appears in a leaf, in left-to-right sorted order.

There are two operations on a B+ tree that make modifies it:

Insertion

- Descend to the leaf where the key fits.
- If the node has an empty space, insert the key into the node.
- *Redistribute Phase*: If the node is already full, split it into two nodes, distributing the keys evenly between the two nodes.
  - If the node is a leaf: take a copy of the minimum value in the second of these two nodes and repeat this insertion algorithm to insert it into the parent node.
  - If the node is a non-leaf: exclude the middle value during the split and repeat this insertion algorithm to insert this excluded value into the parent node.

Deletion

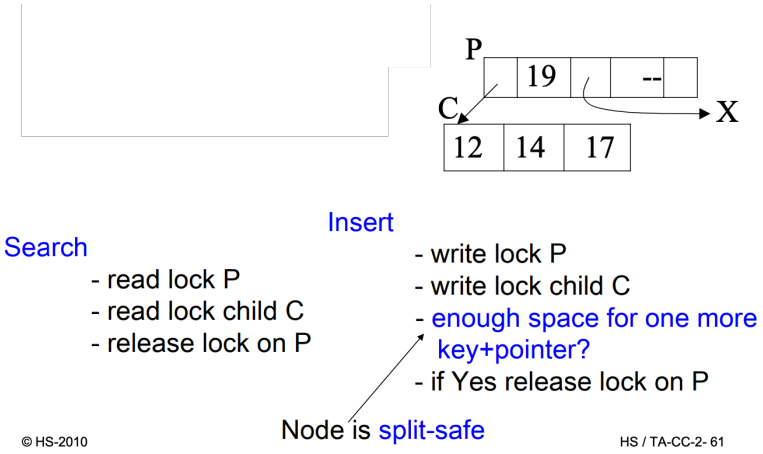
- Remove the required key from the node.
- If the node still has enough keys to satisfy the invariant, stop.
- *Redistribute Step*: If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different "split point" between them; this involves simply changing a key in the levels above, without deletion or insertion.
- *Merge step*: If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invarant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the "split key" from the parent into our merging. In either case, we will need to repeat the removal algorithm on the parent node to remove the "split key" that previously separated these merged nodes - unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

Simple Locking Algorithm
The Simple Locking Algorithm is an example of index locking. We set/remove locks in the following way:

- **Search**: We begin at the root and go down. On each level we rl the child and unlock the parent. This until we reach the leaf.

- **Insert/Delete**: We also begin at the root and go down. On each level we wl the child and then check if it is safe. A node is safe if the changes made will not propagate up beyond the node. In insertions, a node is safe if it is not full. In deletions, a node is safe if it not half empty. If the node is safe, then unlock all of its ancestors.

A con of the Simple Locking Algorithm is that the wl that we put on nodes that are not leafs are unnecessary, because only the leaf nodes are modified. The leaf nodes are the only ones that contain data.



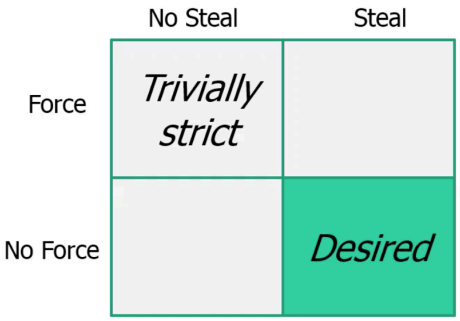
2.7 Recovery Protocols

Read or write operations refer to a page of secondary storage.

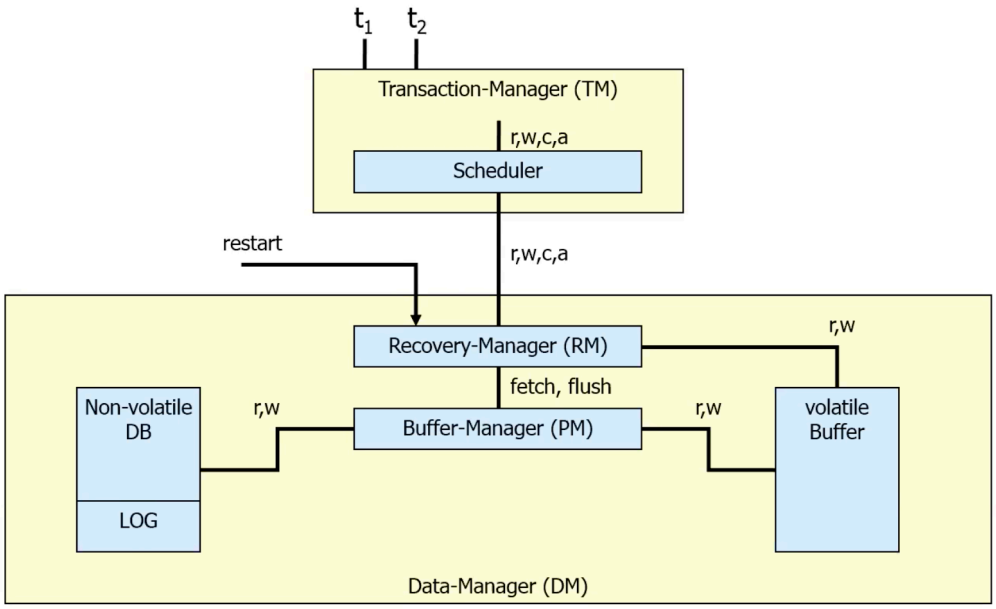
- **Fetch**: (read operation) transfers a page from the database into the buffer, if the corresponding page is not yet in the buffer.
- **Flush**: after a write operation modifies the content of a page, which must be in the buffer; the page can be written to the database (flushed) at once or later.

Theoretically, all changes on objects *o* made by *t* (write operations) should be flushed to disk exactly at commit. Unfortunately this would create a number of problems:

- **Steal** (The risk of Early Disk Writing): usually the operative system and not the database system decides how the pages are used, so the buffer manager might choose to replace the frame in memory which contains the page with the object *o* (i.e. a frame is stolen from *t*).
  In this case things are written on the disk before the commit, which could possibly lead to dirty reads.
- **Force** (What about Late Disk Writing?): It is not optimal to always write on the disk at commit points (force), because this creates a lot of disk access requests at the same time and affects performance. If we allow changes to be flushed after commit (no force), the performance would increase.



Data Manager and Transaction Manager

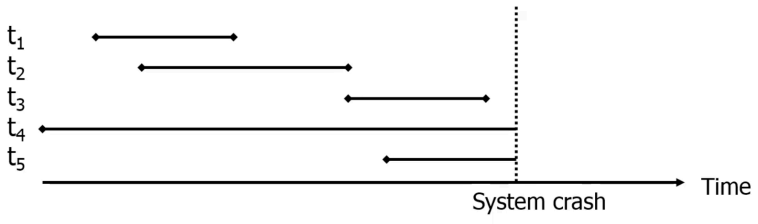


The types of faults, which a DBMS must be able to handle:

1. **Transaction faults**: a transaction does not reach its commit point, e.g. by an error in program or an involvement in a deadlock.
2. **System crash**: parts of (volatile) main memory or buffers get lost, e.g. by errors in DBMS codes, in operating systems or hardwares.
3. **Media fault**: parts of (non-volatile) secondary storage get lost, e.g. by a head crash on a disk, faults in an operating system routine for the writes onto disks.

In the following only fault types (1) and (2) will be considered.

Crash Scenario



Transactions are classified now in two classes:

- Transactions, which were **already released** before the fault. These need a **REDO**, if results are not permanently stored (No- Force situation).
- Transactions, which were **still active** by the time of the fault. These need an **UNDO**, if some results are already stored on disk (Steal situation).

The Recovery Manager (RM) maintains a **log** file :

- If t wants to write a new value of x, a ***Before-Image*** of x is written in the log beforehand (consisting of the ID of t, the ID of x and the old value of x).
- The new value of x is logged in an ***After-Image*** (consisting of IDs for t and x as well as the new value of x).
  To execute a REDO or UNDO of t, the log entries for t are read and processed in reverse sequence. Recovery protocols are classified whether only *After-Images* or only *Before-Images* or both (most systems) are stored.

Any protocol must satisfy the UNDO and REDO rules:

- UNDO-rule („Write-Ahead-Log-Protocol"): The Before-Image of a write operation (the old value of x) must be written into the log, before the new value appears in the stable database.
- REDO-rule(„Commit-rule"): Before a transaction is terminated, every new value that has been written by it must be in the stable storage (in the stable database or in log).

Direct consequence:

- For No-REDO: ensure that all After-Images of a transaction are written in the database before or during the commit.
- For No-UNDO: ensure,that no After-Image of a transaction is written into the database (but only the log) before the commit.

UNDO/REDO Protocol

The RM does not control the buffer manager with respect to the writing of pages into the database. That is, it depends on the page replacement strategies of operating systems, when to execute a flush operation of buffers, e.g., by Demand Paging:

- A write operation writes in (log and) buffer page p
- If p is replaced (written into the database) and the concerning transaction aborts, UNDO is necessary
- If a transaction reaches its commit and p is damaged by a crash before a flush, REDO is necessary

ARIES

Algorithms for Recovery and Isolation Exploiting Semantics. Developed at IBM research in early 1990s. The log record contains the following fields (so that we don't have to copy the whole page into the log):

- LSN (Log Sequence Number, ID for a Log record)
- TransactionID
- Type (Update, Commit, Abort, End (for End of Commit/Abort), CLR)
- pageID
- Offset (indicates where exactly on the page the change happens)

- Length (how many bytes were changed)
- old data
- new data

Steps of the process of recovery after a crash:

1. **Analysis**: Identify dirty pages in the buffer and active transactions at the time of the crash.
2. **Redo**: Repeat all actions from the log, starting from the first action which made a page dirty. Restores the database state to what it was at the time of the crash.
3. **Undo**: Undo transactions that did not commit, so that DB reflects only committed transactions

Three main principles must be followed:

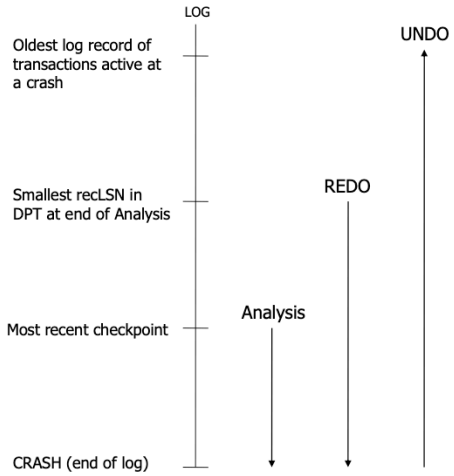
1. **Write-Ahead-Logging** (to ensure Atomicity and Durability) Must force the log record for an update before the corresponding data page gets to disk. Must write all log records for a transaction before commit.
2. **Repeat History During Redo**: repeat ALL actions of the DBMS before a crash, restoring the exact state at the time of the crash.
3. **Log Changes During Undo**: changes made to the database while undoing a transaction are logged to ensure such an action is not repeated in the event of repeated failures/restarts. This information is written into the log in Compensation Log Records (CLRs).

Periodically, the DBMS creates a **checkpoint**, in order to minimize the time taken to recover in the event of a system crash. The following is written to the log:

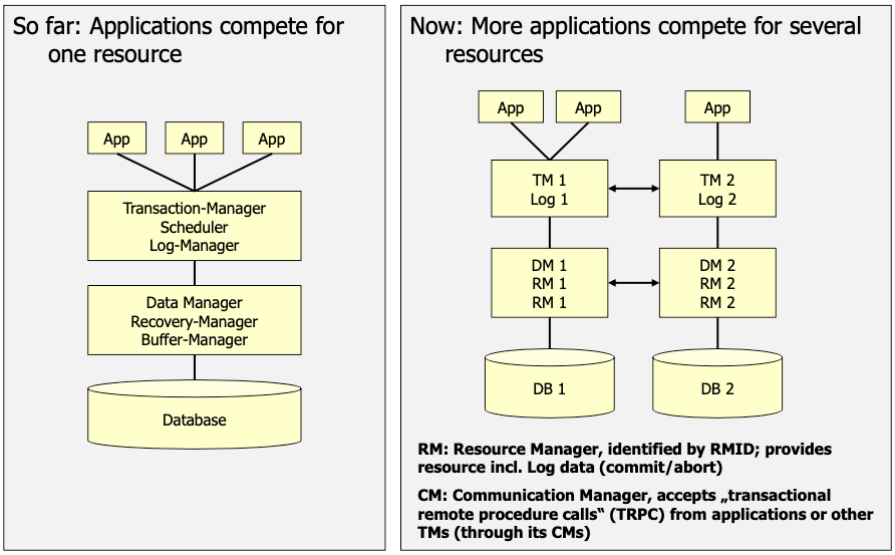
1. **Begin checkpoint record**: Indicates the start of checkpointing
2. **End checkpoint record**: current transaction table and DPT (as at the time of „begin checkpoint“ record is written)
3. **Master Record**: store LSN of the “begin checkpoint” record in a safe place in stable storage.

This is called a „**Fuzzy Checkpoint**“. It is inexpensive, because not all pages in the buffer have to be written to disk. But it is limited by oldest unwritten change in a dirty page. Thus, flushing dirty pages periodically is a good idea. Other transactions can run concurrently to checkpointing.

Example (Crash Recovery Using Fuzzy Checkpoints)



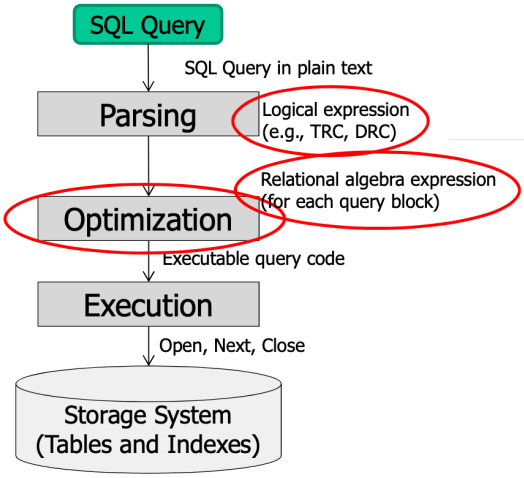
2.8 Distributed Transactions and the CAP Theorem



CAP Theorem

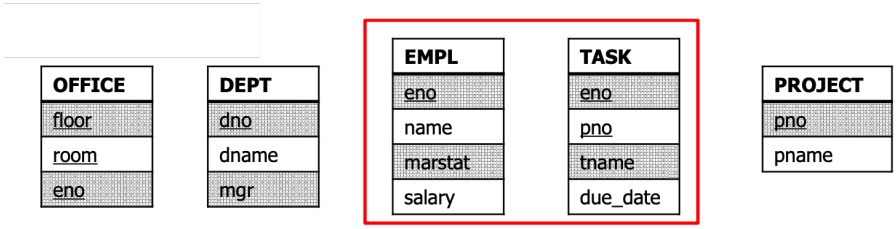
Only two of the following can be achieved together: **C**onsistency, **A**vailability, and Tolerance to Network **P**artitions. Therefor, we must choose between consistency and availability in a network partition.

3. Relational Queries



In this chapter the **query evaluation chain** is introduced and it is discussed how clustering and/or indexing can influence the algorithms. First we present the running example that will be used throughout this chapter.

DB Schema



	Scan of Relation	Equality Selection	Range Selection	Insert	Delete
Heap File	B	0.5 B	B	2	0.5 B + 1
Sorted File	B	$\log_2 B$	$\log_2 B$ + #matching pages	$\log_2 B$ + B	$\log_2 B$ + B
Clustered Tree Index*	0.15 B + 1.5 B	1+ $\log_G 0.15B$	$\log_G 0.15B$ + #matching pages	3+ $\log_G 0.15B$	3+ $\log_G 0.15B$
Unclustered Tree Index*	0.15 B + R·B	1+ $\log_G 0.15B$	$\log_G 0.15B$ + #matching records	3+ $\log_G 0.15B$	3+ $\log_G 0.15B$
Unclustered Hash Index**	B· (R+0.125)	2	B	4	4

- **B:** number of data pages
- **R:** Number of records per page
- **G:** Fan-out of tree index

*:data entry in leaf is 10% of record size, average load per page is 67% ➔ 0.15B leaf pages and 1.5 B pages in clustered heap file

**: avg. load per index page is 80%, 10% data entry ➔ 0.125B pages for data entries

- Sizes:
  - **M** pages in table TASK (1000), p_T tuples per page in table TASK (100). Each tuple is 40 bytes. **m** is the total number of tuples, $m = p_T \cdot M$
  - **N** pages in table EMPL (500), p_E tuples per page in table EMPL (90). Each tuple is 50 bytes. **n** is the total number of tuples, $n = p_E \cdot N$
- Costs:
  - **I/O**: costs for fetching 1 page. Cost Metric: # of IOs to compute operation (e.g. Join)
  - **O-Notation** for complexity of operations
  - No other costs are considered (e.g. for processing of data or data output)

3.1 Implementing Single-Relational Operators

//TODO (CMU lecture 10)

3.2 Join Algorithms

Simple Nested Loop

Cost: $M + (m \cdot N) = M + p_M \cdot M \cdot N = 1000 + 100 \cdot 1000 \cdot 500 = 50.001.000$ I/Os
For every tuple in TASK, it scans EMPL once

```
ANSWER:=[];
FOR EACH t in TASK DO
  FOR EACH e IN EMPL DO
    IF e.salary<40,000 AND e.marstat='single' AND
      t.tname=‘design' AND t.eno=e.eno
    THEN ANSWER:+[<e.name>];
```

Block Nested Loop Join

Cost: $M + (\lceil M/(B-2) \rceil \cdot N)$

Provided we have **B** buffers available. - Use **B-2** buffers for scanning the outer table. - Use one buffer for the inner table, one buffer for storing output.

```
ANSWER:=[];
FOR EACH B - 2 block B_T in TASK DO
  FOR EACH block B_E in EMPL DO
    FOR EACH tuple t in B_T DO
      FOR EACH tuple e IN B_E DO
        IF e.salary<40,000 AND e.marstat='single' AND
          t.tname=‘design' AND t.eno=e.eno
        THEN ANSWER:+[<e.name>];
```

If the outer relation completely fits in memory ($B > M + 2$), then the costs are really low: $M + (\lceil M/M \rceil \cdot N) = M + N$

Index Nested Loop Join

Cost: $M + (m \cdot C)$
C: cost of each index probe, depends on the index type (e.g. B+trees, hashing).

```
ANSWER:=[];
FOR EACH t IN TASK DO
  Lookup t.eno in index on Empl.eno, get tuple e from EMPL
    IF found THEN ANSWER:+[<t,e>];
```

General tips for nested loop joins:

- Pick the smaller table as the outer table.
- Buffer as much of the outer table in memory as possible.
- Loop over the inner table or use an index

Sort-Merge Join

Cost: $MlogM + NlogN + (M + N)$
$(M + N)$ is the merge cost
$MlogM + NlogN$ is the sort cost for the two tables

Phase #1: Sort

- Sort both tables on the join key(s)

Phase #2: Merge

- Scan the two sorted tables with cursors.
  - Advance cursor of T until
    current T-tuple >= current E tuple
  - Advance scan of E until
    current E-tuple >= current T tuple
  - Do this until current T tuple = current E tuple. In that case output matching tuples $< t, e >$ and resume scanning.

Hash Join

Cost: In partitioning phase, read+write both relations; $2(M + N)$. In matching phase, read both relations; $M + N$ I/Os. In total $3(M + N)$.

If tuple $t \in$ TASK and a tuple $e \in$ EMPL satisfy the join condition, then they have the same value for the join attributes. If that value is hashed to some partition i, the TASK tuple must be in t_i and the EMPL tuple in e_i.

Phase #1: Build

- Scan the outer relation and populate a hash table using the hash function h_1 on the join attributes.

Phase #2: Probe

- Scan the inner relation and use h_1 on each tuple to jump to a location in the hash table and find a matching tuple.

```
build hash table HT_R for R
foreach tuple s in S
  output, if h_1(s) in HT_R
```

Join Algorithms: Summary

Algorithm	IO Cost	Example
Simple Nested Loop Join	$M + (m \cdot N)$	1.3 hours
Block nested Loop Join	$M + (m \cdot N)$	50 seconds
Index Nested Loop Join	$M + (M \cdot C)$	Variable
Sort-Merge Join	$M + N +$ (sort cost)	0.59 seconds
Hash Join	$3(M + N)$	0.45 seconds

Hashing is almost always better than sorting for operator execution.

3.3 Tableaus

A tableau is a representation for a special class of conjunctive queries in domain relational calculus (DRC):

$$\{a_1 \ldots a_m | \exists b_1 \ldots b_n (P_1 \wedge P_2 \wedge \ldots \wedge P_k)\}$$

where P_i are atomic predicates (relation predicates or comparisons).

Tableau Method: Example

```
SELECT  c.name FROM EMPL c, DEPT d, EMPL t

WHERE   d.dname='computer' AND c.dno=d.dno AND
        c.marstat='single' AND t.marstat='single' AND
        t.salary<40.000 AND c.eno=t.eno

OR      d.dname='computer' AND c.dno=d.dno AND
        c.marstat='single' AND t.marstat='married' AND
        t.salary<80.000 AND c.eno=t.eno
```

Equivalent query in Domain Relational Calculus (DRC)

{ n | ∃ dname, dno, mst, sal, eno, mgr, n2, mst, sal2, dno2
EMPL(eno,n,mst,sal,dno) ∧ DEPT(dno,dname, mgr) ∧
EMPL(eno,n2,mst2,sal2,dno2) ∧
dname='computer' ∧ mst='single' ∧ mst2='single' ∧ sal2<40.000 } ∪
{ n | ∃ dname, dno, mst, sal, eno, mgr, n2, mst, sal2, dno2
EMPL(eno,n,mst,sal,dno) ∧ DEPT(dno,dname, mgr) ∧
EMPL(eno,n2,mst2,sal2,dno2) ∧
dname='computer' ∧ mst='single' ∧ mst2='married' ∧ sal2<80.000}

For each relation predicate the tableau contains a row, and for each variable a column.

eno	name	marstat	salary	dno	dname	mgr	
	a2						
b1	a2	single	b2	b3			EMPL
				b3	computer	b4	DEPT
b1	b5	single	<40.000	b6			EMPL

Syntactic simplification: Removal of superseded rows

We can replace b5 by a2, b2 by "<40.000" and b6 by b3. We see that the both EMPL rows are the same and we can drop one of them.

eno	name	marstat	salary	dno	dname	mgr	
	a2						
b1	a2	single	b2	b3			EMPL
				b3	computer	b4	DEPT
b1	b5	married	<80.000	b6			EMPL

Semantic constraint propagation ("chase") and deletion of contradictory tableaux.

The rows are contradictory in the marital status. We can drop the whole tableau as the join would be the empty set.

Result tableau

eno	name	marstat	salary	dno	dname	mgr	
	a2						
b1	a2	'single'	<40.000	b3			EMPL
				b3	'computer'	b4	DEPT

{ n | ∃ dname, dno, mst, sal, eno, mgr
EMPL(eno,n,mst,sal,dno) ∧
DEPT(dno,dname, mgr) ∧
dname='computer' ∧ mst='single' ∧ sal<40.000 }

```
SELECT  c.name FROM EMPL c, DEPT d

WHERE   d.dname='computer' AND c.dno=d.dno AND
        c.marstat='single' AND
        c.salary < 40.000
```

Tableau Containment and Equivalence

Definition 3.1 Tableau T_1 is **contained** in tableau T_2 ($T_1 \subseteq T_2$) if

1. T_1, T_2 have the same columns and entries in result rows and
2. The relation computed from T_1 is a subset of the one from T_2 for all valid assignments of relations to rows and for all valid database instances.

Theorem 3.1 (Homomorphism Theorem [Abiteboul et al., 1995])
$T1 \subseteq T2 \Leftrightarrow$ There is a homomorphism h: {variables of T_1} → {variables of T_2} ∪ {constants} with: 1. h(fixed row T_2) = fixed row T_1 2. h(row T_2) = any row of T_1 with the same relation name 3. $h(constant) = constant$ 4. Integrity constraints in T_2 are transferred to the respective symbols in T_1 and are also guarenteed in T_1.

Theorem 3.2
Two tableaux T_x and T_y are equivalent, denoted as $(T_x \equiv T_y) \Leftrightarrow T_x \subseteq T_y \wedge T_y \subseteq T_x$

Example:

$T_1 \subseteq T_2$? → Find mapping h from T_2 to T_1
$T_2 \subseteq T_1$? → Find mapping h from T_1 to T_2

T_1

a		
a	b	(R)
c	d	(R)
e	f	(R)

b<d, d<f

T_2

w		
w	x	(R)
y	z	(R)

x<z

$T_1 \subseteq T_2$, mapping h exists:

h(w) = a
h(y) = c
h(x) = b
h(z) = d

$T_2 \not\subseteq T_1$, because

h(a) = w
h(b) = x
h(c) = y
h(d) = z
but no equivalent for f

$T_3 \subseteq T_4$? → Find mapping h from T_4 to T_3
$T_4 \subseteq T_3$? → Find mapping h from T_3 to T_4

T_3

a		b	
a	c	d	(R)
a	e	f	(R)
g	c	b	(R)
h	e	b	(R)

T_4

a		b	
a	e	f	(R)
h	e	b	(R)

$T_3 \subseteq T_4$
$T_4 \subseteq T_3$

Identity:
h(c) = e
h(d) = f
h(g) = h

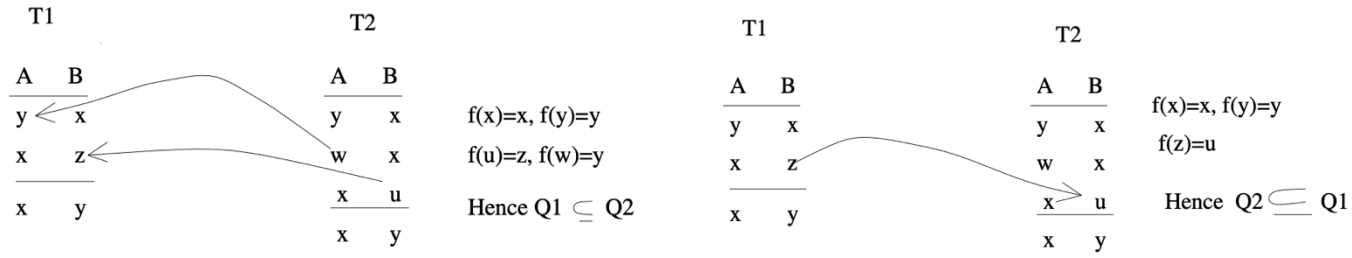


Tableau Minimization

- For each tableau row: delete the row and check equivalence to original tableau
- Unfortunately, this minimization is NP-complete (no problem for small tableaux)

Example:

- Tableau T:

A	B	C
x	4	z_1
x_1	4	z_2
x_1	4	z
x	4	z

- Minimization, step 1: Is there a homomorphism from T to

A	B	C
x_1	4	z_2
x_1	4	z
x	4	z

- Answer: No. For any homomorphism f, $f(x) = x$ (why?), thus the image of the first row is not in the small tableau.

- Step 2: Is T equivalent to

A	B	C
x	4	z_1
x_1	4	z
x	4	z

- Answer: Yes. Homomorphism f: $f(z_2) = z$, all other variables stay the same.
- The new tableau is not equivalent to

A	B	C
x	4	z_1
x	4	z

or

A	B	C
x_1	4	z
x	4	z

- Because $f(x) = x, f(z) = z$, and the image of one of the rows is not present.

- Minimal tableau:

A	B	C
x	4	z_1
x_1	4	z
x	4	z

Theorem 3.3
Every minimal tableau is equivalent to the found tableau (except naming).

- Apply integrity constraints ("knowledge-based optimization" using special reasoners, e.g. chase algorithm) to find minimal equivalent tableau
- Key constraints / functional dependencies: If left sides of FDs (keys) are equal in 2 tableau rows, then right sides are equal as well.
- Referential constraints: "pending" rows can be eliminated.
- Domain constraints: Constant propagation and elimination of unnecessary comparisons

3.4 Tree-Structured Queries

Definition 3.2 Semi-Join
$R \ltimes S = \Pi_R(R \bowtie S)$: The join is projected on the left partner R.
Theorem 3.4
The result of $R \ltimes S$ is a subset of R.
Thus, the result of a semi-join program $(\dots((R \ltimes S) \ltimes T) \ltimes \dots)$ is also a subset of R. That means:
The complexity of a multiway semi-join grows only linearly with the number of semi-joins ($N \cdot k$ instead of N^k) !

Definitions

Range terms "$quant_i r_i \in rel_i$" with $quant_i \in \{\exists, \forall, \_\}$,$rel_i \neq \emptyset$, are represented as nodes k_i.
Dyadic comparison terms $d(r_i, r_j)$ are represented as directed edges $k_i \to k_j$ and labeled with $d(r_i, r_j)$. Is $d(r_i, r_j)$, then the label of the edge is $r_i = r_j$.
The **direction** of edge $k_i \to k_j$ indicates: $SC(r_j) \subseteq SC(r_i)$ i.e. the edge goes from the table that is left on the semijoin to the table that is left e.g. in b) below these semijoins are carried out: $LECTURES \ltimes PROFS$, $PROJECTS \ltimes PROFS$ and $LECTURES \ltimes PROJECTS$

A **path** between two nodes k_i and k_j in the Quant Graph is a sequence of edges connecting the nodes.

- **Predicate path**: if all path edges are labeled.
- **Directed path**: if all pairs of adjacent edges have the same direction (otherwise **undirected**).
- Cycle (**predicate cycle**): undirected path (predicate path) connecting a node with itself.

Quant Graphs

- **Strongly connected**: if for all nodes k_i there is an undirected predicate path to every other node k_j of the graph.

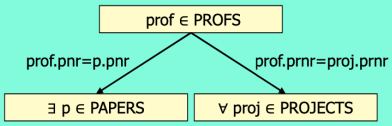
- **Strictly tree-like**: a strongly connected Quant Graph without cycles (can be directly translated into an equivalent semi-join program).

- **Simply tree-like**: a strongly connected Quant Graph without predicate cycles (can often be rewritten into strictly tree-like by exchanging quantifier sequence)

- **Cyclic**: a Quant Graph with at least one predicate cycle (only a few special cases can be rewritten into tree-like structure, usually requires full join execution (exponential in the size of the cycle).

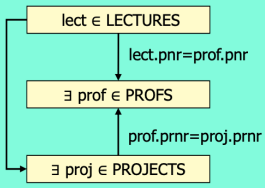
a) Strictly tree-like Quant Graph:

{ prof ∈ PROFS |
∃ p ∈ PAPERS
(prof.pnr=p.pnr) ∧
∀ proj ∈ PROJECTS
(prof.prnr=proj.prnr) }



b) Simply tree-like Quant Graph:

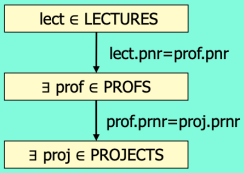
{ lect ∈ LECTURES |
∃ proj ∈ PROJECTS
∃ prof ∈ PROFS
(lect.pnr=prof.pnr ∧
prof.prnr=proj.prnr)}



c) Exchange of quantifiers of the same type in b) leads to an equivalent expression

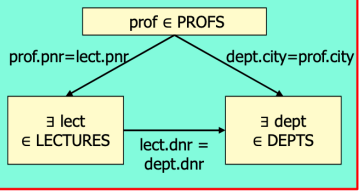
{ lect ∈ LECTURES |
∃ prof ∈ PROFS
∃ proj ∈ PROJECTS
(lect.pnr=prof.pnr ∧
prof.prnr=proj.prnr)}

with strictly tree-like Quant Graph:



d) Cyclic Quant Graph:

{ prof ∈ PROFS |
∃ lect ∈ LECTURES
∃ dept ∈ DEPTS
(prof.pnr=lect.pnr ∧
lect.dnr=dept.dnr ∧
dept.city=prof.city)}



PROFS

pnr	pname	status	city
1	Bolour	assistent	Berkeley
2	Wasserman	tenure	San Francisco

LECTURES

dnr	pnr	room	day	daytime
47	1	502	Tuesday	8:00
20	2	603	Friday	10:00

DEPTS

dnr	dtype	city
47	medicine	San Francisco
20	computer science	Berkeley

3.5 Cost-based Query Optimization

Selinger-style query optimization

Relational Algebra Equivalences

- Selection:
  - $\sigma_{c1} \wedge \ldots \wedge_{cn} (R) = \sigma_{c1}(\ldots(\sigma_{cn}(R)\ldots)$
  - $\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$
- Projection:
  - $\prod_{a1}(R) = \prod_{a1}(\ldots(\prod_{an}(R)\ldots)$
- Join:
  - $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
  - $R \bowtie S = S \bowtie R$
- A projection commutes with a selection that only uses attributes retained by the projection.
- Selection between attributes of the two arguments of a cross-product converts cross-product to a join:

$$\sigma_{R.X=S.Y}(R \times S) = R \underset{R.X=S.Y}{\bowtie} S$$

- A selection just on attributes of R commutes with $R \bowtie S$

$$\sigma_c(R \bowtie S) = \sigma_c(R) \bowtie S$$

- Similarly, if a projection follows a join $R \bowtie S$, we can 'push' it by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.
- For each relation R:
  - $B(R)$: number of pages needed to store relation R.
  - $T(R)$: number of tuples of relation R
- For each attribute a of R:
  - $V(R, a)$: number of distinct values that relation R has in attribute a.

Cost Formulas for Single-relation Queries

Context	Cost
Unique index for equal predicate	$\log_G 0.15B(R)+w$ Note: Only one result tuple!
Applicable* clustered index	$\log_G 0.15B(R)+ F(pred) \cdot B(R) + w \cdot \#tuples$
Applicable non-clustered index	$\log_G 0.15B(R)+ F(pred) \cdot T(R)+ w \cdot \#tuples$
Scan of Relation	$B(R) + w \cdot T(R)$

* **Applicable:** Index I contains column of one or more conjunct in WHERE clause of conjunctive query

Since the CPUs are much faster now than in the 80s when the formulas were thought out, we now set the w (weighing factor) to 0. G is the size of an index page.

Cost Estimation for Join Strategies

Example

```
SELECT e.name, t.tname, e.sal, d.dname
FROM EMP e, DEPT d, TASK t
WHERE t.tname='Design'
AND d.dname='SHIP'
AND e.dno=d.dno
AND e.eno=t.eno
```

DEPT

dno	dname	mgr
50	MFG	5
51	BILL	6
52	SHIP	7

EMPL

eno	name	dno	sal
3	Smith	50	85
4	Jones	50	150
5	Doe	51	95

TASK

eno	tname	pno
3	Design	34
4	Req.	33
5	Impl.	34
3	Design	46

1. Relevant single-relations-access plans:
   - each "interesting" order
   - Other orders if access is cheaper than cheapest "interesting" order

- Scan of each relation
  - EMPL (sorted by eno)
  - DEPT (sorted by dno)
  - TASK (sorted by eno)
- Indexes:
  - EMPL.dno
  - EMPL.eno (clustered)
  - DEPT.dname
  - TASK.eno (clustered)
  - TASK.tname
- Required information
  - Costs (I/O + CPU, but we will just consider I/O costs in the following)
  - Size of result (after selection): #tuples and #pages
  - Order (if any)

Access plan	Selection	Costs	#tuples	#pages	Order
Scan EMPL	-	B(EMPL)	T(EMPL)	B(EMPL)	eno
EMPL.dno	-	0.15*B(EMPL) +T(EMPL)	T(EMPL)	B(EMPL)	dno
EMPL.eno	-	(0.15+1.5)* B(EMPL)	T(EMPL)	B(EMPL)	eno
Scan DEPT	dname= 'SHIP'	B(DEPT)	F_D*T(DEPT)	F_D*B(DEPT)	dno
DEPT.dname	dname= 'SHIP'	0.15* B(DEPT)+ F_D*T(DEPT)	F_D*T(DEPT)	F_D* B(DEPT)	-
Scan TASK	tname= 'Design'	B(TASK)	F_T*T(TASK)	F_T*B(TASK)	eno
TASK.eno	tname= 'Design'	(0.15+1.5)* B(TASK)	F_T*T(TASK)	F_T*B(TASK)	eno
TASK.tname	tname= 'Design'	0.15*B(TASK) + F_T*T(TASK)	F_T*T(TASK)	F_T*B(TASK)	tname

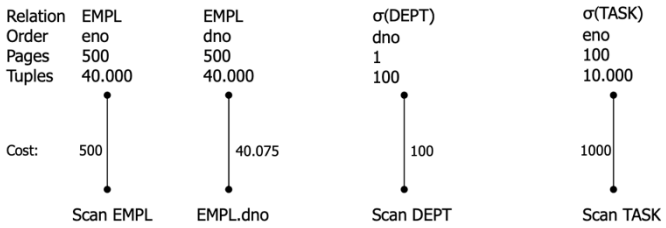
F_T and F_D are the selectivity factors for the selection on TASK and DEPT

B(EMPL)=500, T(EMPL)=40.000
B(TASK)=1000 T(TASK)=100.000
B(DEPT)=100 T(DEPT)=10.000
F_T=0.1; F_D =0.01

Worst case estimation, most likely cheaper in practice

Access plan	Selection	Costs	#tuples	#pages	Order
Scan EMPL	-	500	40.000	500	eno
EMPL.dno	-	40.075	40.000	500	dno
EMPL.eno	-	825	40.000	500	eno
Scan DEPT	dname='SHIP'	100	100	1	dno
DEPT.dname	dname='SHIP'	115	100	1	-
Scan TASK	tname='Design'	1.000	10.000	100	eno
TASK.eno	tname='Design'	1.650	10.000	100	eno
TASK.tname	tname='Design'	10.150	10.000	100	tname

Access plan	Selection	Costs	#tuples	#pages	Order
Scan EMPL	-	500	40.000	500	eno
EMPL.dno	-	40.075	40.000	500	dno
~~EMPL.eno~~	~~-~~	~~825~~	~~40.000~~	~~500~~	~~eno~~
Scan DEPT	dname='SHIP'	100	100	1	dno
~~DEPT.dname~~	~~dname='SHIP'~~	~~115~~	~~100~~	~~1~~	-
Scan TASK	tname='Design'	1.000	10.000	100	eno
~~TASK.eno~~	~~tname='Design'~~	~~1.650~~	~~10.000~~	~~100~~	~~eno~~
~~TASK.tname~~	~~tname='Design'~~	~~10.150~~	~~10.000~~	~~100~~	~~tname~~



2. Compute costs of all relevant join strategies for all remaining pairs of relations (avoiding cross products). Find the best results for
   - EMPL ⋈ TASK
   - EMPL ⋈ DEPT

$$X = \frac{\#tuples * tuple\ size}{page\ size}$$

Access plan	BNL	Merge	#tuples	#pages	Order
Scan EMPL⋈ Scan TASK	B(E)+B(E)* B(T')/(B-2)	B(E)+B(T')	F_{ET}*T(E)*T(T')	X	eno (Merge)
EMPL.dno⋈ Scan TASK	B(E)+B(E)* B(T')/(B-2)	B(E)+B(T')+ B(E)logB(E)	F_{ET}*T(E)*T(T')	X	eno (Merge)
Scan EMPL⋈ Scan DEPT	B(D')+B(D') *B(E)/(B-2)	B(E)+B(D')+ B(E)logB(E)	F_{ED}*T(E)*T(D')	X	dno (Merge)
EMPL.dno⋈ Scan DEPT	B(D')+B(D') *B(E)/(B-2)	B(E)+B(D')	F_{ED}*T(E)*T(D')	X	dno (Merge)

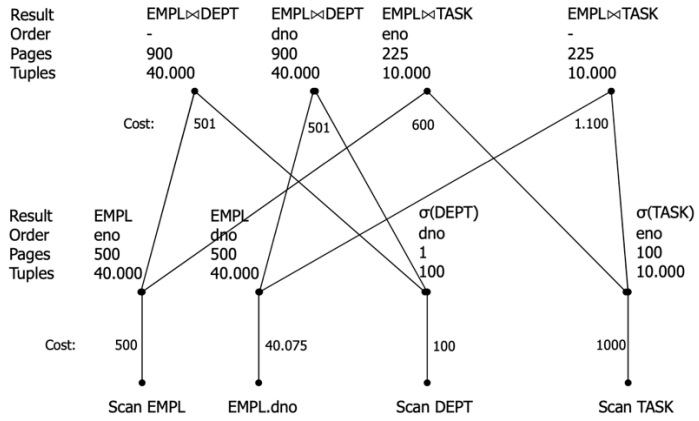
F_{ET} and F_{ED} are the selectivity factors for joins EMPL⋈TASK and EMPL⋈DEPT
T' and D' refer to the relations TASK and DEPT after selection

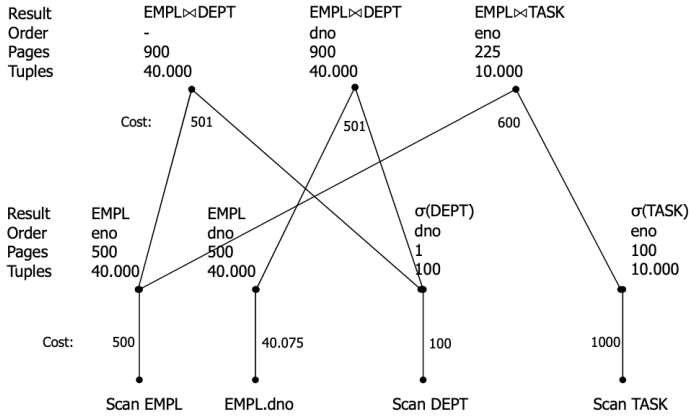
F_{ET}=1/40.000 F_{ED}=1/100 B=52
Assumption: Sorting for Merge join requires only one additional pass

Access plan	BNL	Merge	#tuples	#pages	Order of Best	Sub-plan	Best Total
Scan EMPL⋈ Scan TASK	1.500(1)	600	10.000	225	eno	1.500	2.100
EMPL.dno⋈ Scan TASK	1.500(1)	1.600	10.000	225	-	41.075	42.675
Scan EMPL⋈ Scan DEPT	501	1.501	40.000	900	-	600	1.101
EMPL.dno⋈ Scan DEPT	501	501	40.000	900	dno	40.175	40.676

Plan with BNL join can be removed, because it does not preserve any order and is more expensive than the first plan. Plan with merge join preserves eno order as the first plan, but is also more expensive than the first. Therefore, it will be also removed.

(1) Is 1100 if TASK is the outer relation





Access plan	BNL	Merge	Sub-plan	Best Total
(Scan EMPL⋈$_M$ Scan TASK)⋈ Scan DEPT	226	676	2.100	2.326
(Scan EMPL⋈$_B$ Scan DEPT) ⋈ Scan TASK	1.900	2.800	1.101	3.001
(EMPL.dno⋈$_M$ Scan DEPT) ⋈ Scan TASK	1.900	2.800	40.676	42.576

➔ **and the winner is ...**: Merge join for joining EMPL and TASK (using relation scans), and then BNL join with DEPT

Views and Indexes

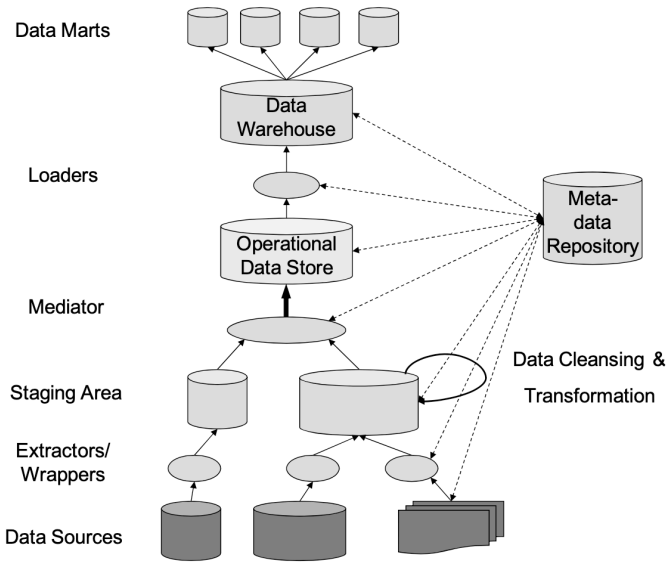
Long term "investment" in access paths. The idea is to store partial results of queries as index or materialized view in the database.

- **Advantage**: Queries can access pre-computed results
- **Disadvantages**:
  - Indexes & Views have to be maintained (higher update costs!)
  - Query processing needs to take into account views & indexes

3.6 Deductive Queries & Integrity Checking

Some queries cannot be expressed in SQL or relational algebra (e.g. Give me a list of all parts that are required to build the component X) (e.g. Give me a list of all known ancestors of „John Doe"). A language that allows **recursion**, such as **Datalog** is required.

Organizations use **Data Warehouses** (DWHs) to analyze current and historical data to identify useful patterns and support business strategies. The emphasis is on complex, interactive, exploratory analytics of very large datasets created by integrating data from across all parts of an enterprise. Data is fairly static. DWHs are focused on queries rather than updates (read rather than write).



Datalog Syntax

Rules are Horn-clauses in the form p:- $r_1 \wedge r_2 \wedge \ldots \wedge r_k$, where p and r_i are relation predicates. e.g.

Semantics: p(X,Y) is true if there exist some X,Y,Z such that every r_i is true

$$p(X,Y) \text{:-} \; r_1(X,Z) \wedge r_2(Z,Y)$$

$$manager(SSN,N) \text{:-} \; empl(SSN,N,M,S,D) \wedge dept(D,DN,SSN)$$

Queries are written either with the head "?-" at the beginning, e.g.

$$?\text{- } empl(SSN, N, \_, \_, D) \wedge dept(D,\text{'computer'}, \_)$$

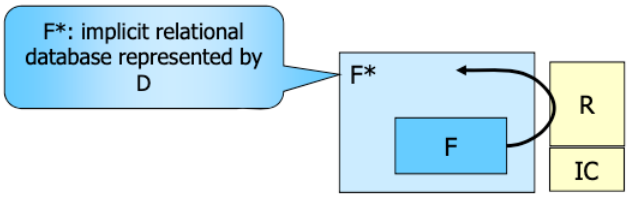
, or with the special query predicate, e.g.

$$q(X,Y)\text{:- } r_1(X,Z) \wedge r_2(Z,Y)$$

Constraints are rules without head that must be always false,
e.g. :- $empl(S, N, M, Salary, D) \wedge Salary < 10.000$

Deductive Databases (DDBs)

A **deductive database** consists of a set F of facts, a set R of deduction rules, a set IC of integrity constraints and the set F* of all explicit and derived (implicit) facts. Deductive Database $D = (F, R, IC)$

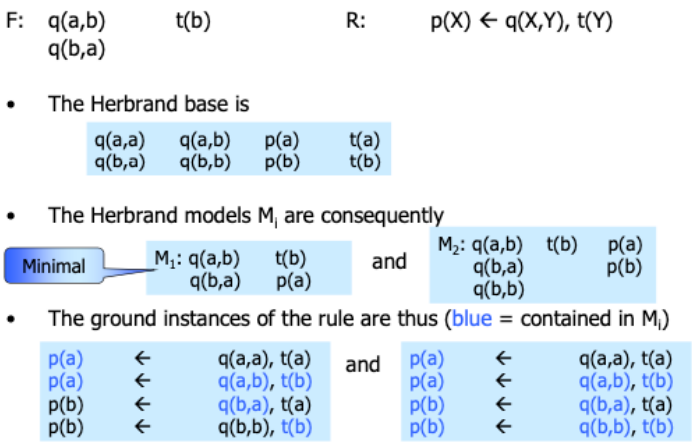


- **EDB**: extensional DB
  - Relations defined as a set of facts in F
  - Base relations
  - Set of facts
- **IDB**: intensional DB
  - Relations defined by rules in R
  - Derived relations

There are two variations of Datalog. **Datalog¬**, where negation is allowed and **NR-Datalog**, where recursion is not allowed.

Herbrand Base of D: all positive ground literals are constructable from predicates in D and constants in D.
Herbrand Model of D: any subset M of the Herbrand Base of D, such that: each fact from F is contained in M. For each ground instance of a rule in D over constants in D, if M contains all literals in the body, then M contains the head as well. A minimal model does not properly contain any other model. F* is formally defined as the minimal Herbrand model of D.

Example (NR-Datalog)



The minimal Herbrand model M_1 contains all the relations in F and p(a) because we can construct it with the rule p(X) ← q(X,Y), t(Y) this way p(a) ← q(a,b), t(b). As you can see, q(a,b) and t(b) are in F.

Only the derivations in blue (in the last bulletpoint) would work.

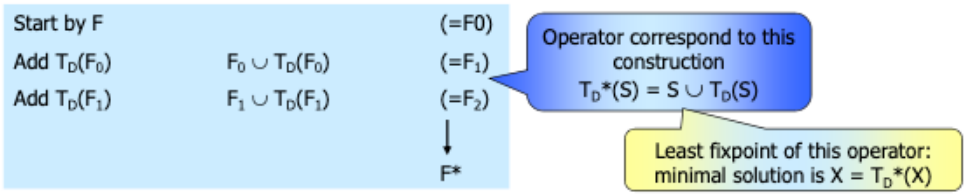
Least Fixpoint for NR-Datalog

F* is created by the repeated (finite) application of the immediate consequence operator T_D (naive evaluation strategies) starting from F results in derivation of implicit facts from F:

$$T_D(T_D(\ldots.(T_D(F))\ldots))$$

For a subset S of a Herbrand Base the application of T_D to S is defined as:

$$T_D(S) := \{H : (H \leftarrow B) \text{ is a ground instance of a rule such that S contains all literals in B}\}$$



Theorem: The uniquely determined least fix point of T_D^* is the minimal Herbrand Model of D.

Example

R
q(X) :- r(X,Y), NOT b(X).
r(X,Y) :- c(X,Y), b(Y).
r(X,Y) :- c(X,Z), r(Z,Y).

F
c(1,2), b(3), c(2,3), b(4), c(1,4).

Solution:

F_0 = {c(1,2), c(2,3), c(1,4), b(3), b(4)} fact base
$T_D(F_0)$={r(2,3),r(1,4)} ∪ $F_0 = F_1$
$T_D(F_1)$ = {q(1), q(2), r(1,3)} ∪ $F_1 = F_2$
$T_D(F_2)$ = {} Fixpoint reached !

Example with negation

If negation occurs in the body of a rule, then there may be no unique minimal Herbrand model any more. Which one is the "natural model" of D, and thus represents the intended semantics of D?

F: q(a,b) q(b,a) t(b)

R: p(X) ← q(X,Y), NOT t(Y)

- The minimal Herbrand models are
  M_1: F ∪ {p(b)}
  M_2: F ∪ {t(a)}

p(b) is derivable from F, M_1 is the "natural" model of D

No reason why t(a) should be true.

- Ground instances of the rules (in M_1, in M_2, in M_1 and M_2):
  p(a) ← q(a,a), NOT t(a)
  p(a) ← q(a,b), NOT t(b)
  p(b) ← q(b,a), NOT t(a)
  p(b) ← q(b,b), NOT t(b)
- Main problem in presence of negation: characterization of the "natural" model of D (representing the intended semantics of D).

Least fixpoint characterization cannot be directly adopted

F: q(a,b) q(b,a) s(b)

R: p(X) ← q(X,Y), NOT t(Y)
t(Y) ← s(Y)

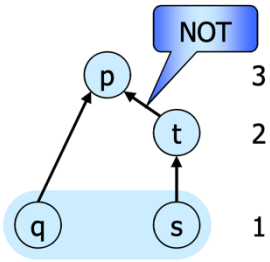
- First application of T_D:
  - No t-fact in F ⇒ p(a) and p(b) are derivable
  - s(b) in F ⇒ t(b) is derivable
- Second application of T_D:
  - No new fact derivable ⇒ fixpoint reached
- It follows therefore:
  least fixpoint of T_D*: F ∪ {t(b), p(a), p(b)} ≠
  "natural" Herbrand model: F ∪ {t(b), p(b)}
- Reason (intuitively): "t(b) arrives too late for preventing derivation of p(a)."

Stratification

The program is **stratified** (or layered) if the predicates "call" each other in a hierarchical order. No two predicates in a layer (stratum) depend negatively on each other. If a predicate p depends on a negative predicate r, then r is in a lower layer. If application of T_D is done layer by layer, the least fixpoint of D is consequently the natural Herbrand model.

It follows therefore:

1. layer: F
2. layer: $F \cup \{t(b)\}$
3. layer: $F \cup \{t(b)\} \cup \{p(b)\}$

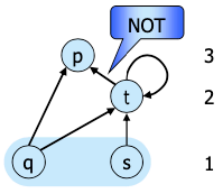


Example Least Fixpoint for Recursive Stratified DATALOG¬ Programs

The recursion leads possibly to more than one application of TD per layer. If negations do not occur in the recursion cycle, there will be no problem.

F:	q(b,c)	s(c)	R:	p(X)	← q(X,Y), NOT t(Y)
	q(b,a)			t(Y)	← q(Y,Z), t(Z)
				t(Y)	← s(Y)

- From this it follows:
  1. layer: F
  2. layer: F ∪ {t(c)}
     F ∪ {t(c)} ∪ {t(b)}
  3. layer: F ∪ {t(c)} ∪ {t(b)} ∪ {p(b)}
     .....

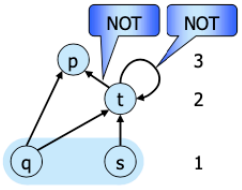


Example Non-stratified DATALOG¬ Programs

Negation in the recursion cycle violates the stratification condition.

F:	q(a,b)	s(c)	R:	p(X)	← q(X,Y), NOT t(Y)
	q(b,a)			t(Y)	← q(Y,Z), NOT t(Z)
				t(Y)	← s(Y)

- Even when applying T_D for t-rules only results in „anomalies“:
  - t(a) is generated, as t(b) is initially missing
  - t(b) is generated, as t(a) is initially missing



- Do t(a) and t(b) belong to the "natural" Herbrand model now? Is there any reasonable "natural" Herbrand model at all?
- t(a) and t(b) are not "natural" consequences of R ∪ F:

t(a)	← q(a,Z), NOT t(Z)		
		b	b
t(b)	← q(b,Z), NOT t(Z)		
		a	a

- Consequently we have two mutually exclusive cases:

either	NOT t(b), consequently t(a)
or	NOT t(a), consequently t(b).

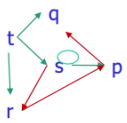
- And thus also two alternative minimal models:

	F ∪ {p(a), t(a)}	, also NOT t(b)
or	F ∪ {p(b), t(b)}	, also NOT t(a).

Are these programs stratisfied?

Program 1

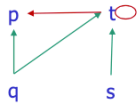
```
q(X)   :- NOT p(X), t(X).
p(X)   :- s(X,X), NOT r(X).
s(X,Y) :- s(Y,X), t(Y).
r(X)   :- t(X), NOT s(X,X).
```



Stratum1: t
Stratum2: s
Stratum3: r
Stratum4: p
Stratum5: q

Program 2

```
p(X) :- q(X,Y), NOT t(Y).
t(Y) :- q(Y,Z), NOT t(Z).
t(Y) :- s(Y).
```



Stratum1: s, q
Stratum2: p
Stratum3: ???

Bottom-Up vs. Top-Down Evaluation

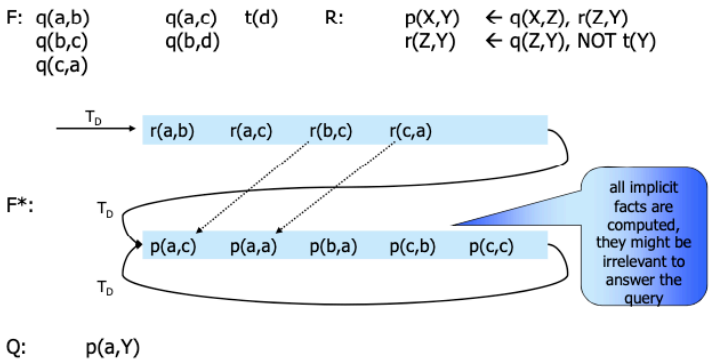
Bottom-Up (Forward Chaining):

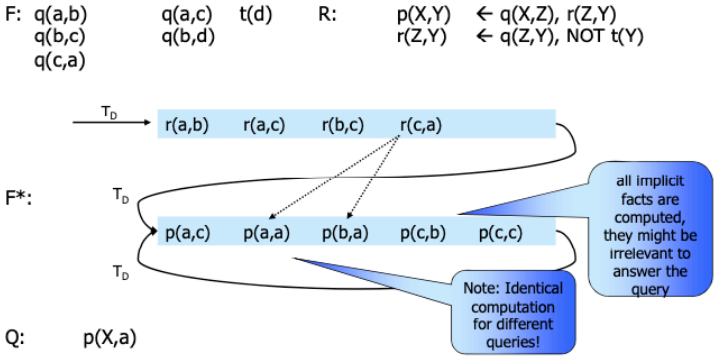
- Generation of implicit facts at evaluation-time.
- Evaluation of the query against temporarily materialized implicit databases. (direct implementation of least fixpoint computation).
- Drawback: when materializing F*, the particular query Q is not considered → many irrelevant answers and intermediate results may be generated.

Top-Down (Backward Evaluation):

- Generation of subqueries until queries to base relations are reached.
- Evaluation of base subqueries against F and upward propagation of answers to the top query.
- As opposed to bottom-up approach: constants in top query and subqueries are passed downwards and provide restrictions while evaluating base queries.
- Drawback: inefficient (or not terminating) for recursive queries.

Example





Integrity Constraints

Integrity constraints (IC) are conditions that have to be satisfied by a database at any point in time (expressing general laws which cannot be used as derivation rules). **Integrity-checking** tests whether a particular update is going to violate any constraint. The main problem with IC-Tests is that a full evaluation of all ICs before every update would be very expensive and would decrease update performance significantly. The solution is to determine a reduced set of simplified ICs for which the checking guarantees satisfaction of all ICs. This approach leads to a specialization of constraints.

Example

inconsistent ← employee(X), works_for(X,X)
(original constraint: no employee works for himself)

insert works_for(john, jim)
delete employee(X) where works_for(X, john)

Constraint Especialization

Input:

- Update U={delete L, insert L}
- Integrity constraints IC (satisfied before the update)

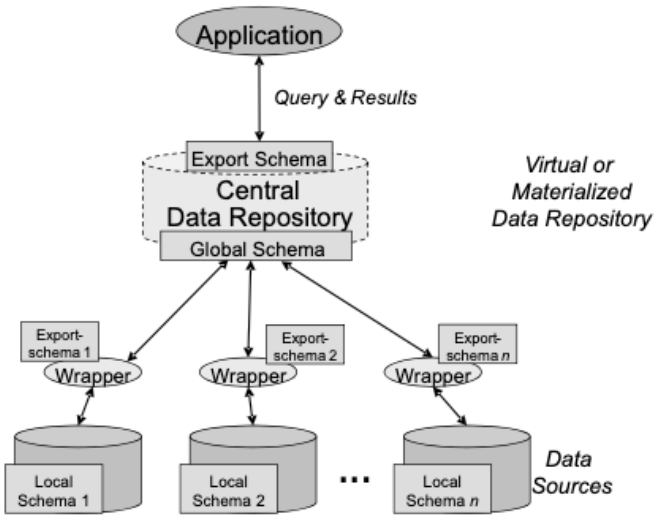
1. IC is affected by U, if IC contains a literal L* that is unifiable with L (resp. NOT L), if U is an insertion (resp. deletion).
2. For every such L, *IC_σ is a relevant instance of IC with respect to U. (where σ is a most-general-unifier of L and L*)
3. Simplified relevant instances of IC with respect to U are obtained by deleting L* from relevant instances.

Example

insert p(a,b) (IC affected) inconsistent ← p(a,b), NOT s(a)
delete p(a,b)
insert s(a)
delete s(a) (IC affected)
inconsistent ← p(a,Y), NOT s(a)

3.7 Querying Data Integration Systems

Data integration is the problem of providing unified and transparent access to a collection of data stored in multiple, autonomous, and heterogeneous data sources



How to specify the **mapping** between the data sources and the global schema?

- **LAV**(local-as-view): The sources are defined in terms of the global schema (i.e. as view on the global schema) – we can thus talk about completeness of sources with respect to global world knowledge
- **GAV** (global-as-view): The global schema is defined in terms of the sources (i.e. as view on the source schemas) – this is more obvious for query processing (view unfolding) but the relationship among the sources (e.g. real-world objects with different ID's in the sources) must be explicitly managed
- **GLAV**(combination of GAV and LAV)

Example

Source Schema S

- em50(Title, Year, Director) *European movies since 1950*
- rv10(Movie, Review) *reviews since 2000*

Global Schema G

- movie(Title, Director, Year)
- ed(Name,Country,Dob)*(European directors)*
- rv(Movie, Review)

Query q

```
SELECT M.title, R.review
FROM Movie M, RV R
WHERE M.title=R.title AND M.year = 2010
```

We have to prove that queries are equivalent or contained in each other:

- **Query Equivalence**: q and q' are equivalent if they produce the same result for all legal databases

- **Query Containment**: q is contained in q' if the result of q is a subset of the result for q' for all legal databases

GAV example

- movie(Title, Director, Year) :- em50(Title, Year, Director).
- ed(Name) :- em50(,,Name).
- rv(Movie, Review) :- rv10(Movie,Review).

Note that the global schema will not know that it cannot find reviews and movies over a certain age

Queries over G can be rewritten as queries over S by unfolding

- q(Title, Review) :- movie(Title, _, 2010), rv(Title,Review).
- ⇒ q'(Title, Review) :- em50(Title, 2010,_), rv10(Title,Review).

LAV example

- em50(Title,Year,Director) :- movie(Title, Director, Year), ed(Director,Country,Dob), Year ≥ 1950.
- rv10(Movie,Review) :- rv(Movie,Review), movie(Movie,Director, Year), Year ≥ 2000.

Here, the view definition explicitly includes the temporal limitations of knowledge about the real world of movies and reviews, thus informative answers to queries outside the boundaries of this knowledge can be given.

The idea is to try to cover the predicates of the query by predicates in the bodies of the source views

- q(Title, Review) :- movie(Title, _, Year), rv(Title,Review), Year = 2010.

The sources defined as materialized views on the global schema:

- em50(Title,Year,Director) :- movie(Title, Director, Year), ed(Director,Country,Dob), Year ≥ 1950.
- rv10(Movie,Review) :- rv(Movie, Review), movie(Movie, Director, Year), Year ≥ 2000.

Answering queries using these views:

- qrewritten(Title, Review) :- rv10(Title,Review), em50(Title,Year,_), Year = 2010.

4. Big Data & Internet Information Systems

4.1 Query Processing in Big Data Systems

Map-Reduce

Map-Reduce is a programming pattern for parallel computation in distributed system. It is defined by two functions:

- **Map(data) → (key,value)**: it reads the input data and emits key-value pairs. Note that the keys are not necessarily unique in emitted pairs.
- **Reduce(key,values) → (key,values)**: it gets input from Map function, which is a set of values for a single key. Note that the input and output structure should be the same and that some implementations require that output is a single value. Furthermore, Reduce can be called multiple times for the same key.

Simple Example (in MongoDB syntax)

CartItem

ID	QTY	PID

Product

ID	Name	Price

Sum of all items in shopping carts, grouped by product ID

```
SELECT PID, SUM(QTY)
FROM CartItem
GROUP BY PID
```

Sum of the value of all items in shopping carts, grouped by product ID

```
SELECT PID, SUM(QTY)*Price
FROM CartItem, Product P
WHERE P.ID=PID
GROUP BY PID
```

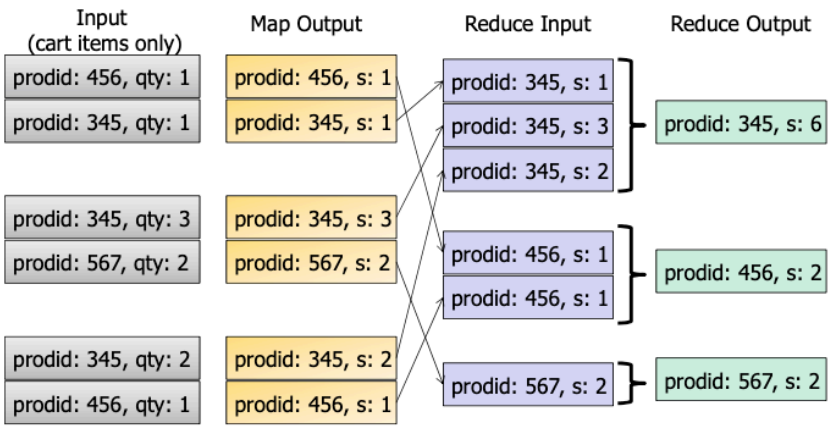
Query: Sum of items in customers' shopping carts, grouped by product ID

Input: Customer collection

```
{
  firstname: "John",
  lastname: "Doe",
  address: { ... },
  cart: [ { prodid: 3456,
            qty: 2},
          { prodid: 6789,
            qty: 1} ]
}
```

```
map=function() {
  if(this.cart!=null) {
    this.cart.forEach(function(item) {
      emit(item.prodid, { sum: item.qty });
    });
  }
}
```

```
reduce=function(key, values) {
  var s=0;
  values.forEach(function(value) {
    s+=value.sum;
  });
  return ( { sum: s} );
}
```



Complex Example (Join between customers and products)

Input: Customers + products collection

```
{
  firstname: "John",
  lastname: "Doe",
  address: { ... },
  cart: [  { prodid: 3456,
             qty: 2},
           { prodid: 6789,
             qty: 1} ]
}
{
  prodid: 3456,
  price: 34
}
{
  prodid: 6789,
  price: 23
}
```

Query: The sum of items and their value in all shopping carts, grouped by product ID

```
map=function() {
  if(this.cart!=null) {
    this.cart.forEach(function(item) {
      emit(item.prodid, { qty: item.qty,
                          price: null,
                          total: null });
    });
  }
  if(this.prodid!=null) {
    emit(this.prodid, {qty: 0,
                       price: this.price,
                       total: 0 });
  }
}
```

Input: Customers + products collection

```
{
  firstname: "John",
  lastname: "Doe",
  address: { ... },
  cart: [  { prodid: 3456,
             qty: 2},
           { prodid: 6789,
             qty: 1} ]
}
{
  prodid: 3456,
  price: 34
}
{
  prodid: 6789,
  price: 23
}
```

Query: The sum of items and their value in all shopping carts, grouped by product ID

```
reduce=function(key, values) {
  var res = { qty : 0, price: null, total: null };
  values.forEach(function(value) {
    res.qty+=value.qty;

    if(res.price!=null) {
      res.total+=res.price*value.qty;
    }

    if(res.price==null && value.price!=null) {
      res.price=value.price;
      res.total=res.price * res.qty;
    }
  });
  return res;
}
```