DATABASE IMPLEMENTATION SUMMARY

Andrea Crotti

17 Febbraio 2010

Contents

1 Possible questions

- B+Trees, operations and advantages
- synchronization problems in transactions (make a list, describe)
- serializability, definitions and analysis of transactions
- failure safety, RC ACA ST and RG
- locking protocols (2PL, S2PL and so on)

- query optimization, implementation of operators and different query representations
- tableaux
- datalog (fix point...)
- distributed database, 2-phase commit, semijoin

2 B+Trees

The *order* of a B+Tree is defined as capacity of the nodes (number of children nodes) in the tree. For example a b+tree of order 2 can have a max of 2 values for every node which mean 3 subpointers.
A particular tree structure where the data is only storead in the leaves. Particulary well suited for search, there also is a link between the leaves.

For example given a possible given a key node of order 2 [A | B] it can have 3 children where:

- $x < A$
- $A <= x < B$
- $x >= B$

2.1 Insertion

- do a search to determine what bucket the new record should go in
- if the bucket is not full, add the record.
- otherwise, split the bucket.
- allocate new leaf and move half the bucket's elements to the new bucket
- insert the new leaf's smallest key and address into the parent.
- if the parent is full, split it also
- now add the middle key to the parent node
- repeat until a parent is found that need not split
- if the root splits, create a new root which has one key and two pointers.

2.2 Characteristics

Given a B+Tree of order b and height h

- max number of records stored: n = b^h (only the leaves count)
- space required to store the tree: $O(n)$
- inserting a record: $O(\log_b n)$
- performing a range query with k elements: $O(log_b n + k)$

3 Transactions

Transactions can be simplified with the **read/write** model, we could have for example those synchronization problems:

- Lost update:
  Losing the effect of an update overwriting it just after
- Dirty read:
  reading a value given by an aborted transaction
- Non repeatable/inconsistent read:
  Two different read values for the same variable in the schedule.
- Phantom problem:
  Insertion or deletions during other actions can create phantoms. This can only be solved by higher order locks.

Thus we need a recovery control and a smart scheduler that satisfies the ACID principles. Serializability and recoverability are two different ways to classify our schedules.

3.1 ACID

Every transaction must be processed in a way that those principles are satisfied.

- Atomicity (a transaction can be rolled back)
- Consistency (database state is always left consistent)
- Isolation (transactions are isolated and don't interfere)
- Durability (data stored in a durable way)

3.2 Read/Write model

A transaction is a finite set of read write operations on objects x, where x is a database object.
D = {x,y,z}
t1 = r1(x), r1(y), w1(z)..

A shuffle is one of the possible permutations of the execution order (also called **complete schedule**).

3.3 Serializability

It holds: $CSR \subset VSR \subset FSR$

3.3.1 Read From

Read From relation is given when $p \rightarrow q$, so the action p is read in q. An action p is *directly useful* ($p \rightarrow q$) if q is read from p or p is read and q consecutive write action.

3.3.2 Alive

An action is *alive* if it's useful for some other actions in the future. For example:
$\{(t_0, x, t_1), (t_1, y, t_2), (t_2, z, t_\infty)\}$ Means that t_2 is alive, t_1 is alive since it's useful for t_2 and t_0 is alive since it's useful for t_1.

3.3.3 Live reads from

Live-reads-from relation is done by all the tuples (t_x, <var>, t_y) which are in a read from relation.
Reads before writing are in relations with $\mathbf{t_0}$ and write before the end are in relation with t_∞.

FSR is the class of all *finite-state-serializable* schedules. But the test for inclusion in FSR has **exponential complexity**, so is not well suitable

3.3.4 Semantics of a schedule

For every schedule we can compute its semantic (the meaning), in short.

- H_s(ri(x)): semantic of the last write
- H_s(wi(x)): semantic given by a function applied on all the previous reads in the transaction on that argument.

The schedule is then a mapping in the form
$H[s] : D \rightarrow HU$ Where HU is the Herbrand universe.

3.3.5 FSR (final state serializability)

Let s and s' be schedules, they are called **finite-state equivalent** if $op(s) = op(s')$ and $H[s] = H[s']$ are valid. Therefore $s \equiv s'$

We can't determine it by simply the last state but also by previous write operations, also the previous write operations must be taken into account.

3.3.6 VSR (View state serializability)

A serial schedule is schedulable if it is equivalent to serializable schedule such that.
RF(s') = RF(s3), op(s') = op(s3) are holding.

To find out if $s \in VSR$ we need to compute the *Read From Relations* between transactions and see if they are conflicting.

For example given the transaction:
s3 = $w_2(y)$ $r_3(y)$ $w_3(z)$ $r_2(z)$ $r_2(x)$ $w_1(y)$ $w_1(z)$ c_1 c_2 c_4 The READ-FROM relations are:
$(t0, y, t3), (t2, z, t3), (t2, x, t0), (t1, y, t_\infty), (t1, z, t_\infty)$

3.3.7 CSR (conflict state serializability)

Two actions are in **conflict** if they operate on the same object and at least one is a write.

- r1(x) w1(x) $\rightarrow$ conflict
- w1(x) w2(x) $\rightarrow$ conflict

s and s' are **conflict equivalent** if

- $op(s) = op(s')$
- $conf(s) = conf(s')$

They are *conflict-equivalent* if they can be turned one into the other by a sequence of non conflicting swaps of adjacent actions.

A schedule is *conflict serializable* if it is conflict-equivalent to a serializable schedule (swapping all the actions should lead to a serializable schedule). Testing for membership to CSR can be done in polynomial time.

For conflict and view serializability checking just take into account the transactions that actually commmits and **sort out** the aborted.

- Precedence graphs
  $T1 <_s T2$ if there are actions A_1 of T_1 and A_2 of T_2 such that:
  - A_1 is ahead of A_2 in S
  - A_1 and A_2 involve the same element
  - At least one is a write action

  With those information we can write a graph and graph is **cyclic** then the schedule is not conflict serializable.

4 Transaction recovery

Serializability does not avoid synchronization problems between processes. Recovery properties are **orthogonal** to serializability.

Under which conditions a schedule allows a **correct recovery of transactions**?
We need to be able to make sure we can go back to the starting point from an aborted transaction (for example).

In order of strictness we have
$RIGOROUS \subset STRICT \subset ACA \subset REC$ Strict schedules solve WW and WR conflicts, while Rigorous schedules also solve the RW conflict (less dangerous).

4.1 Recoverability

Every transaction will not be released, until all the other transactions from which it has read, are released RC is the class of all recoverable schedules.

In other words we can say that if we read from another transaction we must make sure that the other transaction does the commit before us. For example.

- $s_1 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ w_1(z)\ c1 \notin RC$
- $s_2 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c1\ c2 \in RC$

Because t_2 reads from t_1 on y and c2 commits before c1.

4.2 Avoidance of cascading aborts

Recoverability does not suffice in some situations, because the values restored after an abort, may be different from the before image.
So we need to rollback aborted transactions **and** redo committed transactions. A transaction is in ACA when is only allowed to read values from already successfully completed transactions.

- $s_1 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2 \notin ACA$
- $s_2 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c1\ r_2(y)\ w_2(y)\ c_2 \in ACA$

  Here t_2 tries to read y before t_1 (who wrote on it) has committed.

4.3 Strictness

A schedule is *strict*, if an object is not read or overwritten, until the transaction, which has written it at last, is terminated.

- $s_1 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2 \notin ST$
- $s_2 = w_1(x)\ w_1(y)\ r_2(u)\ w_1(z)\ c_1\ w_2(x)\ r_2(y)\ w_2(y)\ c_2 \in ST$

Here t2 was trying to write on x before t1 (who wrote x for last) terminated. Another way to check is checking that after every write operation the corresponding transaction is terminated.

4.4 Rigorous schedules

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

5 Concurrency Control Protocols

Techiques thanks to which the DBMS can generate correct schedules. They can use locking mechanism or not.

We must consider

- Safety
- expressiveness

5.1 Locking scheduler

Applied for synchronization of accesses on same data objects. For a schedule s a DT(s) it the projection of s on the actions of type "a,c,r,w". (removing the locking and unlocking operations).

- rl(x) read lock
- wl(x) write lock

In general unlocks don't have to be done immediately after but they must be not redundant.

5.1.1 Two phase locking (2PL)

A locking protocol is *two phase* if:

- In the first phase of the transaction locks will only be set, in the second phase will only be removed.
- Locks must be removed as soon as possible.

PROS:

- easy to implement
- good performances
- easy to distribute

CONS:

- **not** deadlock free
- transactions may starve!

It's proved that:
$\epsilon(2PL) \subseteq CSR$
Other possible variants of 2PL are:

- Conservative 2PL: All locks available since BOT
- S2PL: All write locks hold till EOT (removing locks just after the transaction is concluded)
- SS2PL: All locks hold till EOT (too restrictive, transactions should be too short in this case)

Removing a lock is always safer at the end of the transaction, but usually much earlier. $\epsilon(S2PL) \subseteq CSR \bigcap ST$, the S2PL scheduler is safe.

Trick:
Once you have set up a write lock you can also read directly.

- Disadvantages of 2PL
  - big locking objects: a few locks but with many conflicts
  - small locking objects: more concurrency but a higher cost
- Example
  When no other locks can be set and you can't go forward you get a **deadlock**. In that case you have to abort the transaction with the smallest index and restart it only when other interfering transactions have committed. Given s:
  s = $r_1(x)$ $w_1(x)$ $r_3(x)$ $w_3(x)$ c_3 $r_2(x)$ $w_2(x)$ $w_2(x)$ c_2 a_1 We have the sequence
  - $rl_1(x)$
  - $r_1(x)$
  - $rl_2(x)$
  - $r_2(x)$
  - **a1**, here ew can't set a write lock on x and if we unlock first we can't set other locks later
  - $wl_2(x)$, we go directly after the first commit, and can't unlock before c_2 is committed
  - $w_2(x)$
  - c_2
  - $wu_2(x)$, finally unlocking it
  - $wl_3(x)$
  - $w_3(x)$
  - c3
  - $wu_3(x)$
  - $wl_4(x)$
  - $w_4(x)$, unlock only after c_4 is committed
  - c_4

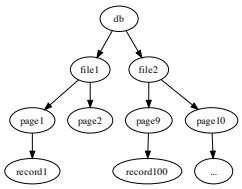
- $wu_4(x)$
- $\mathbf{rl_1(x)}$, here finally we have the restart of the aborted transaction
- $r_1(x)$
- $wl_1(x)$
- $w_1(x)$
- c1
- $w_{u1}(x)$

5.1.2 MGL

We need **intentional locks**. The idea is for a transaction to indicate, along the path, what locks will require in some of the possible paths.

- irl: a read lock will be requested
- iwl: a write lock will be requested
- riwl: current node is read locked but also a write lock will be requested later in the subtree.

Considering a database structure like:



We can set up locks with a higher granularity on one particular subtree.

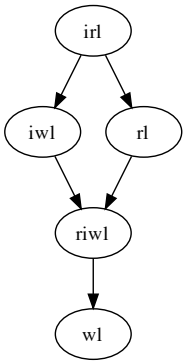
Read lock is also called *shared* lock, while write lock is *exclusive*. To be able to apply locks on one tree we must first have acquired an *intentional lock*. And you can't remove an intentional lock until you have a lock on one child node, the locks are set top-down and removed bottom-up.

Each transaction is locked/unlocked as follows:

1. x not root, t_i must own a ir- or iw- lock on the parent node to set a rl- or irl- lock
2. x not root, t_i must own a iw lock on the parent to be able to set a wl- or iwl- lock
3. to read (write) x, t_i must own a r- or w-lock on a child of x

4. t_i can't remove an intentional lock on x, as long as t_i has still a lock on a child of x.

A transaction can only hold **one** lock on an object, this are the possible updates



- Compatibility table

	rl	wl	irl	iwl	riwl
rl	+	-	+	-	-
wl	-	-	-	-	-
irl	+	-	+	+	+
iwl	-	-	+	+	-
riwl	-	-	+	-	-

5.1.3 Index locking

Assuming insertions also S2PL could fail (phantom problem for example). Conflict serializability is only guaranteed as long as we don't add objects.

- no index (disable completely insertions)
- index on fields which are used in those transactions (which normally at run time is not known anyway)

5.1.4 Predicate locking

Only lock on all records satisfying some logical predicates (not commonly used as it's too much expansive to implement).

5.1.5 Locking in B+Trees

In B+Trees real data is only contained in the leaf nodes, no information given by the intermediates. A node is safe if changes will not propagate to higher levels of the tree

- **insertions**: Node is not full
- **deletions**: Node is not half empty

Working on the B+Trees involves:

- Searching
  - go down from root
  - read lock child, unlock parent
- Insert / delete
  - go down from root
  - write lock child, then check if safe

The problem is that there are two many useless write locks, since the data is only phisically stored in the leaves.

Improved tree locking: Try to lock only the leaf, if not safe backtrack to root and use previous algorithm.

Another possible way could be to use MGL, but deadlocks are possible.

5.1.6 Non locking

Other possible ways without locking are possible

- Optimistic CC Use private copies and if there is a conflict abort and restart
- Timestamp based CC Every TA gets a timestamp, if p_i and q_i are in conflict execute p_i before q_i, so it generates conflict serializable schedules. It's not more efficient but can be used in distributed systems.

5.2 Concurrency control in SQL

SQL allows to set up different security levels, for different usages:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

In order of safety and decreasing concurrency allowed. Normally serializable is always used since it's the only one avoiding all the synchronization problems.

6 Recovery protocols

We need to be able to recover from transactions faults.

- REDO if transaction was done but not stored
- UNDO if transaction was partially written before the fault

A recovery manager get's the transactions from the scheduler and take some precautions before actually loading them.

Write a new value of x:

- store a *Before-image* of x ({ID, x, oldx})
- store an *After-image* of x ({ID, x, newx})

We could also avoid UNDO and REDO if we put some constraints on the execution of read/write in the system.

- UNDO-rule: (write-ahead log protocol) before image of a write operation must be written to the log **before** the new value appear stable in the database
- REDO-rule: (commit-rule) before a transaction is terminated, every new value written by must be in the stable storage.

6.1 Steal and force

- Steal: Replace the frame in memory which contains the page with the object o (the frame is stolen).
- Force: When the transaction commit, we ensure that all the changes to the object are immediately **forced** to disk.

Best combination is **Steal** + **no force**.

7 ARIES (*Algorithm for Recovery and Isolation Exploting Semantics*)

Steal-no force approach used.

- Write-ahead-logging
- repeat history during redo repeat ALL actions before the crash
- logging changes during undo write in the **CLRs** changes made during undoing.

Goals of ARIES are:

- Simplicity
- Operation logging
- Flexible storage management
- Partial rollbacks
- Recovery independence
- Logical undo
- Parallelism and fast recovery
- Minimal overhead

7.1 WAL

- force log record update **before** corresponding data gets written to disk
- write all records for a transaction **before commit**

7.2 LOG

The log must contain every information useful for reconstructing the correct values. In particular

- LSN (log sequence number, for every log record)
- old data
- new data

... In plus we must keep a

- **transaction table** (one entry for each active transaction and a lastLSN)
- **dirty page table** (one entry per dirty page in buffer and a reclLSN, the log record who first caused the problem)

Redo is done from the reclLSN and undo until lastLSN.

7.3 Checkpointing

Periodically a **checkpoint** is created by the DBMS to minimize the time needed to recover.
Store also the LSN of the checkpoint on disk. You must clear the dirty page table before doing it, and then the analysis phase can start from the last checkpoint created.

7.4 Recovery

A nice thing about ARIES is that it works even if we have a failure during a recovery.

7.4.1 Analysis

- Reconstruct state at checkpoint (using the record)
- Scan log forward from checkpoint
  - End record: remove transaction from transaction table
  - Other records: Add transaction to transaction table, set lastLSN=LSN, change status to commit
  - Update record: if P not in DPT, add P to DPT, set reclLSN=LSN.

This phase is used to determine:

- point where to start the REDO pass (reclLSN)
- the *dirty pages* at moment of crash
- *transactions active* at the moment of crash

7.4.2 REDO phase

- repeat history to reconstruct state at crash (reapply all updates)
- reapply logged actions

Redo redoes all changes to any page that was dirty at the moment of crash

7.4.3 UNDO phase

Undoes all the transactions that were active (but didn't commit) at the moment of crash.

8 Indexing

Indexes are used to speed up the retrieval of records in response to certain search conditions. *Any field* could be used to construct the index.

Three kinds of indexes are:

- primary (used on ordering fields)
- secondary
- clustering

Index can also be **dense** or **sparse**, depending by the number of entries that it has for *every search key value*.

8.1 Clustering

A clustering index instead does not have one entry for every possible value, but it points to a file which contains all the records where the field has that value. In this case records are phisically ordered, so we can have some problems in insertion / deletion, that's why we normally reserve one entire block for *each value* of the clustering field.

8.2 Secondary

A **secondary index** provides a secondary means of accessing a file for which some primary access already exists. It's useful to work on an arbitrary number of tuples since otherwise we should search in linear time.

8.3 Multilevel indexes

8.3.1 Multilevel indexes using B-Trees and B^+-Trees

9 Query evaluation

In genernal a select would be translated to an innested loop, possible ways to improve:

- selection before join (makes the tables to join smaller)
- semi joins
- index, hashing
- sequence optimization (change the order of operations)

Other possible ways are:

- sort/merge algorithm
- note/join
- hash join

9.1 Parameters

A pass in query evaluation is a read of the data to be processed. To evaluate how many passes are needed we have to consider:

- buffer size
- various indexes available
- data distribution

Some operators can be done on the fly and the performances highly depend by the order of execution and the indexing

9.1.1 One pass

- Selection always in one pass
- Projection only if buffer is big enough (in memory sorting and duplicates elimination)
- Join (if smaller relation fits in buffer then nested loop)

9.1.2 Two passes

Two pass are usually enough for anything, partitioning into acceptable size, sort and hash.

9.2 Access planning

The access plan is important to get the maximum possible speed.

- Join sequence
- Join implementation
- Parallelism
- Distribution

Dynamic programming techniques are used to find the best tradeoff. Cost estimation is important in finding the right access planning.

- intermediate result sizes
- phisical access dependencies

9.3 Cost estimation

We need to compute the selectivity of an operator

- **monadic** operators: divide output size by input size
- **dyadic** operators: divide output size by product of the inputs

$V(R, y)$ represents the number of different values for attribute y in the relation R.

- equality: $F = \frac{1}{V(R,y)}$

- range selection:
  - $\sigma_{y>k}$: $F = \frac{max(R,y)-k}{max(R,y)-min(R,y)}$
  - $\sigma_{y<k}$: $F = \frac{k-min(R,y)}{max(R,y)-min(R,y)}$
- join (on 2 relations): $\frac{1}{max(V(S,y),V(R,y))}$

9.3.1 Estimation rate

Given one theoretical estimation (for example using standard distribution) and a concrete estimation we can compute the accuracy with. $\frac{|est_1-est_2|}{est_2} \times 100$ %

9.4 Query representation

- Tuple relational calculus
- Relational algebra
- Domain calculus
- DPNF

9.4.1 Tuple relational calculus

A query is in the form: {<goal list> OF EACH $r_1 \ldots$, EACH r_n in R_n: *selection predicate}* We can have different possibilities for selecting, from normal boolean conditions to join on other lists for some attributes. *SOME* and *ALL* are the quantifiers used for testing.

Example:

```
[<e.name OF EACH e IN EMPL:
         ALL e IN EMPL (e.salary < 40000 OR e.dno != d.dno)]
```

9.4.2 Domain relational calculus

Domain variables $x_i \in Dom$ represents attributes. Predicates:

- Atomic predicates
- $\neg A$
- $\forall x_i A$

- $\exists x_i A$

Facts: Atomic predicates with possible universal quantified variables
Rules: Disjunctive Horn clauses.

Example: {Ename $\in NameType|$
$Eno \in Enotype$,
$Sal \in \{10000..90000\}$,
$Dno \in Dnotype$,
$Mgr \in Enotype(EMPL(Eno, Ename, `single', Sal, Dno) \wedge$
$DEPT(Dno, `computer', Mgr) \wedge$
$Sal < 40000)\}$

9.4.3 Relational algebra

A *relational complete* language is able to express all queries expressible by RA.

We define other operators

- Projection
- Selection
- Join
- Union
- Intersection
- Difference

For example, names of the dependents with one son and salary > 10000:

$\pi_{name}((\sigma_{sons=1}SONS) \bowtie (\sigma_{salary>10000}DEPS))$
Moreover we have defined:

- cartesian product (R $\times$ S)
- natural join (equal in their common attribute name)
- semi-join (there is A couple in S with the equal attribute)
- θ-join, equi-join (join with a condition of = or <> on one attribute)

9.4.4 Tableaux calculus

There is a close relation with Domain calculus and tableau representation. A tableaux can be optimized finding the minimal outcome of all equivalent tableaux.

$T_1 \subseteq T_2$ if

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

We just need to find a function mapping from one to the other.

Minimization: Delete every row and check equivalence with the original tableau. (NP-complete procedure)

9.5 Implementation of relational operators

Relational operators are implemented to be as fast as possible using buffers and other available structures given by the DBMS.

9.5.1 Summary table

- N : pages external relation
- M : pages internal relation
- B : # of buffers available
- pr : number of tuples in one page

General costs for various opertions:

METHOD	COST
GENERAL MERGE SORT	$2N \cdot (1 + \log_{B-1}(\frac{N}{B}))$
SELECTION(no index)	$O(M)$
SELECTION(no index, sorted)	$O(\log_2 M)$
PROJECTION (sorting)	$O(M \log M)$
PROJECTION (hashing)	$B - 1$ partitions
JOIN (simple nested)	$M + pr \cdot M \cdot N$
JOIN (page nested)	$M + M \cdot N$
JOIN (block nested)	$M + (\frac{M}{B-2}) \cdot N$
JOIN (index nested)	$M + ((M \cdot pr) \cdot$ cost finding S
HASH JOIN	$3(M + N)$
SORT MERGE	$M \log M + N \log N + (M + N)$

9.5.2 Sorting

- Normally data are requested in order
- Sorting is useful for building B+tree index
- Useful to eliminate duplicates

The problem is sorting X Mb of data in Y Mb of ram (X » Y)
2 way sorting:

- Pass 1: reads a page, sort it, write it (only 1 buffer needed)
- Pass 2,3...: three buffer pages are used

It's a *divide et impera* algorithm, we sort the smaller parts and then merge them together while we go on. 2 buffers for reading the sorted data and writing them in the third buffer.

In general a **n-way sorting** can use more buffers and produce directly more.

Number of passes needed for N records and B buffers needed can be computed as: $\#pass = \lceil \log_B N \rceil$ Using B+trees for sorting is a good idea only if they are clustered.

9.5.3 Computing costs

I/O cost for fetching 1 page. M denotes the number of pages No other costs are considered.

9.5.4 Selection

Cost of selection can be seen as **No index, unsorted**:

- scan entire relation, O(M)

No index, sorted

- binary search, $O(\log_2 M)$

Using an index instead can be much faster

- clustered/unclustered index

9.5.5 Projection

1. remove unwanted duplicates
2. Eliminated duplicate tuples

Sorting should be used, first sort and then remove duplicates in the same run $Cost = O(M \log M)$

A modified external merge sort is normally used, in step 0 we can directly remove duplicates. Another possible way is to use hashing for the projection, applying recursively when needed.

In general projection is always *sorting-based*, because in hash case we could:

- fail because bucket too big
- hashing is sensitive to data distribution
- sorting makes the output sorted
- already specialized code exists

9.5.6 Join

Given the query $R \bowtie_{i=j} S$ Easiest way is to use a simple nested loop algorithm, very inefficient in many cases. For each tuple in R we must scan the inner tuples in S. So it's important to put **bigger relations in the outer loop**.

Page oriented nested loop: For each page of R, we get each page of S, and write out the matching tuples on <r,s> where r is in R-page and S in S-page.

Index nested loop join: Put an index on the join column of one relation and make it the inner relation (to exploit the speed). The cost of finding the tuple depends on the height of the B+tree.

Block oriented nested loop: Cost: Scan of outer + #outer blocks * scan of inner

Sort merge join:

- Sort R and S on the join column, then do a merge while keeping both tables aligned

So R is scanned once, each S is scanned once per matched tuple. So we get $M \log M + N \log N + (M + N)$ In practice the cost is linear.

Hash join: Partition both relations using hash function h. Read in a partition of R, hash it using h2 (different hashing functions).

- In partitioning phase: read write both 2(M + N)
- In matching phase: read both (M + N)

$3(M + N)$

9.5.7 Set operations

Intersection and cross-product are special cases of join. We can use a *sorting based approach* (a simple sort and merge) or a *hash based approach* (hash function on tables and union discarding duplicates) to union.

9.5.8 Aggregate relations

In general they require scanning the relation entirely.

9.6 Query optimization

$Query \rightarrow parsing \rightarrow optimization \rightarrow execution \rightarrow storagesystem$ In general we use heuristics to find the best access plan

9.6.1 Join ordering

"Database the complete book" contains many informations about it. Join cost depends on the order, but the final result will always be the same (for associativity of join operator). In join we should use the smaller relation as outer relation, indexing on the inner relation can exploit it.

Most DBMS chose a left-deep plan, associating to the left.

9.6.2 Possible heuristics

1. size of intermediate relations
2. selections should be pushed down in the tree (even if in some cases it's not better)
3. most restrictive joins first
4. postpone joining of large relations

10 Deductive database

10.1 Intro

Some queries can't be expressed by SQL or RA, for example:

- Give me a list of all known ancestors of "John Doe"

(Recursion is needed in those cases) Prolog allow function symbols as argument of predicate while datalog does not.

Example:

```
empl(1, 'jim', 100).
empl(2, 'brad', 200).

manager(1, 10).
manager(2, 10).

%% find the employers with the same manager
same_manager :- empl(X, _, _), empl(Y, _, _), X /= Y, manager(X, Z), manager(Y, Z).
```

10.2 Definition

A deductive database consists of

- Facts F
- Rules R
- Integrity constraints IC
- Explicit and implicit (derived) facts F*

D = (F, R)

Then we have to consider separately the cases in which negation/recursion are allowed/not allowed.

10.3 Herbrand model

F* is the minimal Herbrand model of D. A minimal model does not properly contain any other model.

10.4 F* derivation in NR-datalog

The minimal Herbrand model is obtained through repeated application of T_D starting from F using rules R.
$T_D(T_D..(F))$ F* is created by *naive* application of this procedure.

10.5 F* derivation in DATALOG¬programs

Recursion in this case make it possible to apply more than once T_D per layer. Allowing negation in datalog programs can lead to not unique *minimal Herbrand models*. Given for example $p(X) : -s(X,Y), NOTt(Y)$ Doesn't specify how Y could be, and everything different from a constant where t holds could generate a minimal Herbrand model.

10.6 Elements

The elements are

- Rules (which are **horn clauses**)
- Queries
- Constraints (also facts, that are true)

Variables can be free or bounded. P(X, Y) :- R(X, Z), R(Y,Z). Here X is bound but Z is free.

- Theory: schema + integrity constraints
- Interpretation: database state

10.7 Intentional database

Intentional database (IDB) can be defined by a system of algebraic equations where:

- variables for each relation respectively names of predicates
- Translate rule bodies into algebraic expressions
- All rules for a derivable relation are summarized by unions.

10.8 Semantics of a deductive database

A deductive database D = (F, R) F* is the implicit relation.

- Is F* uniquely determinable?
- What meaning of *derivable*?

F* is the **minimal Herbrand Model** of D.

10.9 Possible evaluation strategies

- Backward/derivation/top down (as in prolog): efficient selection (unification) but possibly not terminating
- Forward/generation/bottom up (databases): Finite sequence of algebraic manipulations, but possibly large unnecessary results

10.10 Bottom up

Generate facts at evaluation time, it's a direct implementation of least fixed point computation. Many irrelevant facts are also generated, because we generate **all** possible facts and then set up connections only on some of them.

A possible improvement would be to generate a meta relation query_ which reflects top-down processing.

10.11 Top down

Generation of subquerys until facts are reached (proving or disproving), may not terminate. Backtracking to choice point gives the final answer.

A way to make top-down approach faster would be to store subqueries and answers (dynamic programming approach).

10.12 Integrity constraints

Are conditions that have to be satisfied by the database at any point in time.

11 Internet and database systems

11.1 Services of a distributed system

- remote database access

- distribution transparency
- concurrency control and recovery
- homogeneous/heterogeneous systems

RDA defines interoperability of systems, allowing different systems to co-operate on the same database remotely. RDA defines how the final server/client connection happens.

Transaction management in a distributed system is more complicated, every RM must satisfy a DO-UNDO-REDO protocol.

- DO: execute operations and records them in log file
- UNDO: reconstruct object given the log file
- REDO: reconstruct object from old object state and the log entry

11.1.1 Two-phase commit

- Phase1: ask participators of a transaction whether they agree on a commit
- Phase2: execute commit if all participators agree.
- when they've all agreed the transaction is called "prepared".

Possible failing reasons:

- confirm of a "cancel" operation
- integrity tests, if failing the commit is denied.

Commits can be *eager* or *lazy*, in case of lazy commits a crash could still violate the "durability" in ACID.

- 2-phase commit in the distributed case
  It has distributed transaction managers and managed by a central coordinator. Distributed commit:
  - localprepare
  - distribute prepare
  - commit step
  - complete step

  We also need a query coordinator that tells when the commits are feasible.