Exam notes

Functional dependencies

> Trivial FD: $X \to Y$ is trivial if: $Y \subset X$.
> Non-trivial FD: *see above*.

> Superkey: Superset of a candidate key.

> Candidate key: Minimal set of a key.

> Primary key: Candidate key chosen by designer.

> Prime attributes: Attributes contained in candidate key.

> Closure of **FDs**: Set of all dependencies that include *F* and all those that can be inferred from *F*.
> Denoted as F^+.

> Closure of **set of attributes**: Maximum set of attributes that can be inferred from the provided set of attributes.

Normalization

Database design principle to

1. *reduce redundancy*
2. *avoid complexities*
3. *organize data in a consistent way.*
4. *eliminate duplicates*
   but loses out on

- query efficiency

> 1NF: No multivalued attributes, i.e. only simple values.

> 2NF: Parts of the candidate key must not be functional dependent on non-prime attributes.

- *Example:* $\{\mathbf{A}, \mathbf{B}\}$: $\overbrace{\mathbf{A} \to C, D, E}^{\text{illegal}}$
  $\Rightarrow$ No arrows from part of candidate key!

Decompose: *Make new table containing LHS and RHS. Remove RHS from origin table.*

> 3NF: Non-prime attributes must not be functionally determined by other non-prime attributes.

- *Equivalent to no transitive dependencies*: $\mathbf{A} \to \overbrace{B \to C}^{\text{illegal}}$
  $\Rightarrow$ No arrows starting from non-prime attributes!

Decompose: *Make new table containing LHS and RHS. Remove RHS from origin table.*

> BCNF: No dependencies from non-key attributes to key attributes.

- *Example*: $\{\mathbf{A}, \mathbf{B}\} \rightarrow \overbrace{C \rightarrow \mathbf{A}}^{\text{illegal}}$
  $\Rightarrow$ No arrows pointing towards prime attributes!

Decompose: *Remove one of the prime-attributes*!
NB: BCNF is lossless, but it is not dependency preserving!

> Losslessness: Decomposition is lossless if we can recover initial table by performing multiple joins.
> Dependency preservation: We do not lose dependencies. True if all dependencies can be inferred from the current set.

Triggers

- Can not modify the database schema; solely DML (*Data Manipulation Language*) for data-level operations.

> Invoked *automatically*.
> Defined in terms of the event *that invokes it*, and the *action it performs*.
> May operate either `BEFORE` or `AFTER` the execution of the event that invokes it.

Storage

Main memory vs. disk:

- Data access from disk is typically 2 orders of magnitude slower.
- Data in main memory is **volatile**, while **non-volatile** for disk.
- Disk is solely mechanical moving parts, while main memory is completely electronic.

> Each time we read from disk, we retrieve a *block* of records.
> Size of these *blocks* is fixed, but depends on the OS.

Indexes

> Index: data structure that facilitates quicker access to a data.

- *can be* used for both main memory and disk!
- *stored in a data file*.

> Primary index: Indexes on the primary key (ordered on key).

- → Dense index: Has exactly one index entry for each search key value.
- → Sparse index: Has fewer index entries than search key values.

> Clustering index: Indexes on a non-key field (ordered on field).

- → One index entry of each (distinct!) value of the field.
- → Each index points to first data block of records for search key.

> Secondary index: Not ordered on the index's search key (purely a mess!)

> Multi-level index: Structure of index on index until all entries of the top-level structure fit into 1 disk block.

- → Pin top-level index in main memory (RAM).

B^+-trees

- *multi-leveled* indexing structure
- tailored for disk-based data organization: *aligns with disk block sizes, so very efficient for disk storage and access.*
- grows horizontally by splitting the root.

⇓ Operation	Average	Worst case
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

> Balanced: all paths from root to a leaf have the same length.

- → guarantees good search performance!

Authorization

Preparing SQL statements they are compiled **only once**, but **executed multiple times**!

> SQL injection attacks: Using dynamic SQL queries with user inputs, people can inject malicious statements!

- ***Fix***: Bind inputs to variables and then sanitize them!

  > Sanitize: Filters matching the input to patterns and stripping it from invalid patterns.

The Database Administrator (***DBA***) can grant permissions:

```
GRANT *privileges* ON *object* TO *identity* WITH GRANT OPTION;
```

> Transitive granting: Allows users to grant other users privileges: `WITH GRANT OPTION`.

There are the following *privileges*:
`SELECT`:
`INSERT`:
`DELETE`:
`UPDATE`:
`REFERENCES`: *the right to include an attribute in a foreign key*.

Revoking privileges can be done using

```
REVOKE *privileges* ON *object* FROM *identities* <CASCADE/RESTRICT>
```

where `<CASCADE/RESTRICT>` revolves around *transitive granted privileges*.

To create roles

```
CREATE ROLE Paymaster;
GRANT UPDATE(salary) ON Payroll TO Paymaster;
GRANT Paymaster TO amoeller WITH GRANT OPTION;
```

NoSQL

≡ *Not Only SQL*.
Motivation

- Huge amounts of data
- High performance (response time)
- Consistency less important
- Scalability (relational model *too restrictive*!)
  ⇒ Often done using **distributed databases**!

Data-model: "*Master-slave replication or master-master replication and sharding*."
Commonly *hashing* and *range partitions* is used to localize data.

Categories of systems:

1. Document-based: ***MongoDB***
- → Normalized: *Decompose into several documents of similar structure and content*
- → Denormalized: *All information in one document*
1. Key-value stores: ***Voldemort*** (builds on ***DynamoDB***)
2. Column-based / wide-column: ***Apache HBase***
3. Graph-based: ***Neo4j***

> CRUD: **C**reate, **R**ead, **U**pdate, **D**elete (+ **S**earch) operations provided through API.

> CAP properties: Goals when replicating data across distributed network:

- **C**: Consistency (same value among replicas)
- **A**: Availability (successful read/writes)
- **P**: Partition tolerance (robust under network failure)
  Note: In DBMS *consistency* was integrity constraints, now we only want identical replicas!

> CAP theorem: Impossible to guarantee all three: C, A, P.

- → NoSQL systems slack on **C**: consistency.

Keywords:

- *No schema required*, instead *semi-structured* data such as JSON, XML!
- *Less powerful query languages*!
- May provide *multiple version storage*.

- *Vertically* scalable (not horizontally)

Recovery strategies

> Idempotent: *If a system crashes during the recovery stage, the new recovery must still give correct results*.

> Cascading rollback: *A transaction fails or aborts, and as a result, all other transactions that read data modified by the failed transaction must also be rolled back.*

> Checkpointing: process where the system periodically writes all modified (*dirty*) pages from memory to disk.

- improves **recovery** efficiency
- ensures a **consistent** state can be restored after unexpected crash.

> Shadow-paging: copy-on-write technique for avoiding in-place updates of pages

- when a page is modified (*dirty*), the system writes changes to a **new (shadow) page** instead of overwriting the old.
- upon commit the page table pointer is switched to the new page (**atomic**!).
  ≈ no-undo / no-redo

⟹ *possible to use both techniques, but often redundant because both handle recovery well*.

> Write ahead logging (*WAL*): maintain “before image” in log (main-memory!), flush to disk before overwritten with “after image” on disk

> Undo: rollback changes of *uncommited* transactions
> Redo: reapply changes of *commited* transactions after a crash.

> Steal: *uncommited* dirty pages can be written, so you need to ***undo*** them.

- e.g. to save space in main memory you flush the dirty pages more often.

  > Force: *commited* pages are immediately written (forced!) to disk, so you do not need to repeat them (***no-redo***).

- generally not favorable because it leads to a ton of continuously costly I/Os

	Steal	**No-steal**
Force	Undo / no-redo	No-undo / no-redo
No-force	Undo / redo	No-undo / redo

> Undo/redo:

1. Undo all transactions that has a log entry of *"start"* but no *"commit"*.
2. Redo all transactions that has a long entry of *"start"* and *"commit"*.

> ARIES: *Recovery algorithm implemented in many IBM related databases*.

Based on 3 key ideas:

1. Write Ahead Logging: *Log changes **before** writing them to disk.*
2. Repeat history during redo: *Reapply all actions, including uncommited, to **reconstruct state***
3. Logging changes during undo: *When undoing, **log each undo action**, so crash during recovery is safe* (**idempotent**)

Using a 3-phase recovery:

1. Analysis: *Identify dirty pages and active transactions*.
2. Redo: *Repeat history*.
3. Undo: *Roll back any uncommited transactions using the log*.

Transactions

- Multiuser DBMS systems on single-threaded CPUs are interleaved.
- Provides mechanism for logical units of database processing.

Database access operations: Done by reading database item into *a program variable*, and then writing the *program variable value* to a database item.

> Def. = (***Transaction***): *A unit of work defined by a* `BEGIN TRANSCATION` and `END TRANSACTION`.

> Isolation level: Defines degree to which transactions are *isolated*!

1. `SERLIAZABLE`: guarantees transactions behave as though they were serial.
2. `REPEATABLE READ`: read and write locks on rows (not on ranges)!
   → no dirty reads!
   → *phantoms*: new rows can still be inserted
3. `READ COMMITED`: read and write locks on rows, but read locks released immediately after reading.
   → no dirty reads, but because read locks are released you can get different values.
4. `READ UNCOMMITED`: no read locks!
   → very risky!

(!) Isolation levels are a *personal choice*, which affects how *you* see the database state!

Isolation level (highest to lowest!)	Dirty reads allowed?	Unrepeatable reads allowed?	Phantom reads allowed?	Concurrency
`SERIALIZABLE`	No	No	No	Low
`REPEATABLE READ`	No	No	Yes	Medium
`READ COMMITED`	No	Yes	Yes	High
`READ UNCOMMITED`	Yes	Yes	Yes	Very High

> Overview of problems:

1. ***Dirty reads***:
   - A transaction updates a database item and then fails for some reason.
   - Updated item is accessed by another transaction before it is changed back to its original value.

2. ***Unrepeatable reads***:
   - A transaction reads the same item twice.
   - Another transaction changes the value between first and second read
3. ***Lost update***:
   - Two transactions access same item rendering its value incorrect.
4. ***Phantom reads***:
   - A transaction queries for an item twice.
   - Another transaction inserts an item matching the query between first and second read.

Concurrency control

> Schedule: Order in which transactions or operations are performed.

- $\longrightarrow$ *Example:* for

$$\begin{aligned} T_1: &\quad R_1(X), W_1(X) \\ T_2: &\quad R_2(X) \end{aligned}$$

```
can be interleaved in many different ways!
```

$$\begin{aligned} 1: &\quad R_1(X), W_1(X), R_2(X) &&\Rightarrow \text{Serial} \\ 2: &\quad R_2(X), R_1(X), W_1(X) &&\Rightarrow \text{Serial} \\ 3: &\quad R_1(X), R_2(X), W_1(X) &&\Rightarrow \text{Concurrent} \end{aligned}$$

- $\longrightarrow$ Relevant for: *consistency* and *isolation*!

> Conflict: Occurs when you have

1. actions from at least 2 transactions
2. one of them is a write
3. they are on the same attribute

> Conflict equivalent: If transactions I and J do not conflict, we may swap their order to produce a new schedule S'. We then denote S and S' are **conflict equivalent**.

- *Note*: conflicting operations must maintain their relative order!

> Serializable: A transaction that is **(conflict) equivalent** to a serial schedule.

> Precedence graphs:
> 4. For every transaction make a node.
> 5. Draw directed edge from i to j if

$$\begin{aligned} RW: &\ R_i(X), \ldots, W_j(X) \\ WR: &\ W_i(X), \ldots, R_j(X) \\ WW: &\ W_i(X), \ldots, W_j(X) \end{aligned}$$

3. If there is a cycle (despite labeling of attributes!) then it is NOT serializable.
4. To obtain the serializable schedule, follow the arrows from end to end. Note; there may be several.

There are different types of schedules:

1. Cascadeless schedule: Mitigates cascading rollbacks

> "*Every transaction reads only items written by committed transactions*"

2. Strict schedule: Mitigates cascading rollbacks

> "*Transaction can neither read or write item until the last transaction that wrote it has committed.*"
> ⇒ *Transactions can only read committed values!*

3. Recoverable schedule:

> "*No transaction commits until all transactions that have written an item that it read have committed*"

Locking

> Binary locks (*mutex locks*): Ensures safe access to data by having

- (1) Locked: *data is not accessible by other*
- (0) Unlocked: *data is available for access*
  Important: Once locked, no other transaction can read or write the data item!
  This is different than shared- and exclusive locks.
  Description:

  A lock needs to be acquired before any read or write, and must unlock again after completing all operations; it unlocks only items it currently has a lock on, and requests locks only for items it currently has no lock on.

> Shared- and exclusive locks (*multiple-mode locking*):

- Shared lock (S): *Allows multiple transactions to read the data item.*
  - *Note*: Shared locks are unique for transactions and hence are not "shared"!
- Exclusive lock (X): *Allows one transaction to write (and also read, if needed)*.
- Unlock (U): ...
- Lock-conversion: *A transaction can upgrade a shared- to an exclusive lock, or downgrade an exclusive- to a shared lock.*
  Description:

  A transaction must acquire a shared or exclusive lock prior to reading, and an exclusive lock prior to writing, and it must unlock again after all its operations are completed; it issues lock requests only for items it does not already hold a lock on, and it issues unlock requests only for items it holds a lock on.

> 2-phase locking (*2PL*) :

1. Phase 1 (*growing phase*) :
   - Transaction *may request* locks
   - Transactions *may not unlock* locks
   - (Can *convert (upgrade)*: $S(X) \rightarrow E(X)$)
2. Phase 2 (*shrinking phase*) :
   - Transactions *may not request* locks
   - Transactions *may unlock* locks
   - (Can *convert (downgrade)*: $E(X) \rightarrow S(X)$)

(!) When the first locks is released, the transaction moves from phase 1 to phase 2.
$\Longrightarrow$

> Deadlocks: Cycle of transactions all waiting for another to unlock a data item.

> Validation (optimistic) concurrency control: Do the work on *local copies only*, before committing check if there is any issues; if so abort and restart, otherwise write changes to database.

1. ***Read phase***:
   - Read and write operations are made in local workspace (*copy of relevant data only*!)
2. ***Validation phase***: (check serializability!)
   - Assign timestamps when starting validation, check for R/W, W/W conflicts from *older* transactions.
3. ***Write phase***:
   - Local changes are written to database if validation is successful.

Validation is based on 3 steps: $\text{For all other recent: } \mathrm{TS}(\mathrm{T_j}) < \mathrm{TS}(\mathrm{T_i})$

1. $\mathrm{T_j}$ executes all 3 phases before $\mathrm{T_i}$ begins (*serial execution*!)
2. $\mathrm{T_j}$ completes its write phase before $\mathrm{T_i}$ starts its write phase, and does not change any items read.

$$\mathrm{write\_set}(\mathrm{T_j}) \cap \mathrm{read\_set}(\mathrm{T_i}) = \emptyset$$

3. $\mathrm{T_j}$ completes its read phase before $\mathrm{T_i}$ starts its read phase, and does not change any item that is read or written to.

$$\mathrm{write\_set}(\mathrm{T_j}) \cap \mathrm{read\_set}(\mathrm{T_i}) = \emptyset$$
$$\mathrm{write\_set}(\mathrm{T_j}) \cap \mathrm{write\_set}(\mathrm{T_i}) = \emptyset$$

   $\Rightarrow$ and one of these steps must be true!

> ==Multi-version 2-phase locking w. certify locks ==: Allows a transaction T' to read a data item X while it is write-locked by a conflicting transaction T.

- $\Rightarrow$ 3 locks: *read*, *write* (now not exclusive, reads possible), and *certify* (fully exclusive).

1. When T wants to write X, it creates a second version X' <u>after</u> obtaining a *write*-lock.
2. Other transactions continue reading X.
3. When T is ready to commit, it obtains a *certify*-lock on X'.
4. Committed version X becomes X'.
5. T releases *certify*-lock.
   Core idea:
   Instead of just read and write (shared and exclusive, respectively), we now have three locks, read, write, certify (shared, shared with reads, exclusive, respectively). With this, we can allow some reads while an item is write locked, but transactions may have be delay when waiting to certify and deadlocks may occur.

> Snapshot isolation: Each transaction sees a snapshot of the database at the time it started.

- $\Rightarrow$ *No read locks, only write locks*!

Granularity locking

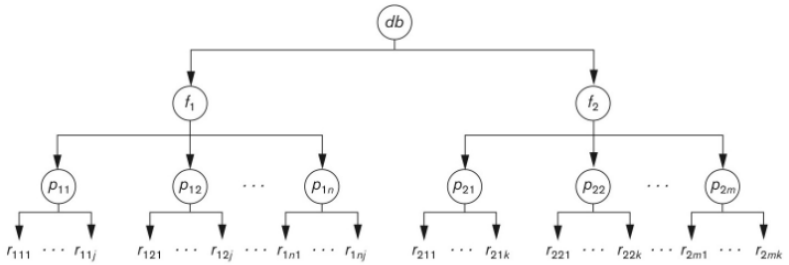
> Def. (*Granularity*): Size of a lockable unit of data

- $\longrightarrow$ *coarse*: entire database

- $\rightarrow$ *fine*: tuple or attribute of relation
  $\Longrightarrow$ *significantly affects concurrency performance*:
- low degree of concurrency for coarse granularity!
- high degree of concurrency for fine granularity!

> **Rule of thumb**:
> "*We want to choose granularity to reflect the typical access size of transactions*"

> Granularity hierarchy: B-tree structure of coarse and finer granularity:



Core idea: Writing to B-tree family indexes could lock a lot of pages, as all index access starts at the root. A more optimistic approach holds shared locks on non-leaf nodes, and exclusive locks on the leaf, unless a split becomes necessary.

To manage this hierarchy we introduce 3 more locks:

- ***Intention-shared*** (*IS*): indicates that a shared lock(s) will be requested on some descendant nodes(s)
- ***Intention-exclusive*** (*IX*): indicates that an exclusive lock(s) will be requested on some descendant node(s)
- ***Shared-intention-exclusive*** (*SIX*): indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendant nodes(s)
  $\Rightarrow$ *typically when you want to read a node and want an exclusive lock on a descendant node*.

Timestamping

- Every transaction gets a *timestamp*; $\mathrm{ST}_1 \rightarrow \mathrm{TS}(\mathrm{T}_1) = 1$.
- Resolves *at what point in time* the transactions should see the state of information in.

> Single version:

1. $R(X)$: $W_{TS}(X) \leq TS(T_i)$
- *If the write-timestamp is younger than yours, then you should not be allowed to read.*
  $\Rightarrow$ if allowed: $R_{TS}(X) = \max(R_{TS}(X), TS(T_i)$
  $\Rightarrow$ else abort and rollback.
2. $W(X)$: $W_{TS}(X) \leq TS(T_i)$ AND $R_{TS}(X) \leq TS(T_i)$
- *Same as before, but now also if someone younger has written the value, then you should not be allowed to change it at an earlier time.*
  $\Rightarrow$ if allowed: $W_{TS}(X) = \max(W_{TS}(X), TS(T_i))$
  $\Rightarrow$ else abort and rollback.

> Multi version: Reads will always be okay, but not necessarily for writes.

1. $R(X)$: always allowed, but read version with highest W_{TS} that is $\leq TS(T_i)$
2. $W(X)$: find version with highest W_{TS} that is still $\leq TS(T_i)$.
   If $R_{TS} \leq TS(T_i)$, then it is allowed. Then make a new version with $R_{TS} = W_{TS} = TS(T_i)$.
   (!) Storage (RAM and disk) is required to maintain multiple versions!

Relational algebra

- Set of (*basic*!) operations for the *relational model*.
- Working with mathematical sets, duplicates are automatically eliminated!
- Operations are nested in sequences.
- Can not compute recursive closures! $\Rightarrow$ <u>not</u> Turing complete.

`SELECTION`, $\sigma_{\text{Condition}}(\text{Relation})$; specify a subset of of tuples that satisfy a condition.

- *Resulting schema is unchanged*!

`PROJECTION`, $\pi_{\text{attributes: } a_1,\ldots}(\text{Relation})$; specify a subset of attributes.

`RENAMING`, $\rho_{a_1 \to b_1}(\text{Relation})$; rename relation or attributes.

`CARTESIAN PRODUCT`, $R \times S$; cross join all tuples that can be constructed by combining two tuples from either relation.

- *Essentially just adding the relations together*.

`THETA JOIN`, $R \bowtie_\theta S \equiv \sigma_\theta(R \times S)$; combine tuples through θ-condition from two relations into single, "longer" tuples.

$\Rightarrow$ *Variations of join*:

- `EQUIJOIN`, ...; only $=$ comparison used, and combines pairs of attributes that have identical values in every tuple.
- `NATURAL JOIN`, $R * S$; equijoin on attributes of the same name.
- `OUTER JOIN`, $R \bowtie, \bowtie, \bowtie S$; keep all tuples in R, or all those in S, or all those in both relations regardless of whether they have matching tuples in the other relation.

Aggregate functions and grouping; `SUM`, `AVERAGE`, `MINIMUM`, ...
$_{\text{group attributes}}\Im_{\text{aggregate functions}}(\text{Relation})$

Set operations

- Arguments must have *the same schema*!
- Result will again have *the same schema*!

`DIVISION`, $A \div B$; return tuples from A which match all tuples from B in all attributes.

- *Result schema is* $\text{Attributes of } A - \text{Attributes of } B = X$

`UNION`, $R \cup S$; combines tuples of the two relations and will contain all tuples in either or both.

`INTERSECTION`, $R \cap S$; combines all tuples that are in both relations

`DIFFERENCE`, $R - S$; all tuples that are in R but not in S.

Relational calculus

- *Identical expressive power as relational algebra*.
- *Two types of calculi; **TRC** and **DRC***: both have same *expressive power*!

(!) *Query languages* are based on **relational calculus***, whilst their *implementations* are based on **relational algebra**. This is because **relational calculus** is:

1. declarative: *describes what you want, <u>not</u> how to get it*

2. non-procedural: *does not specify the steps to compute the result*

Tuple relational calculus

- Expresses results as *sets of tuples* that satisfy *a condition*.

$$\{ t \mid \text{Relation}(t) \wedge (\text{ Conditions}(t)) \}$$

Conditions are *boolean expressions*, consisting of operations

$=$	$\neq$	$\leq, \geq$	$<, >$

and connected through logical operators

$\wedge$	$\vee$	$\neg$
"and"	"or"	"not"

Also with quantifiers

$\forall$	$\exists$	$\Rightarrow$	$\neg\exists$	$\neg\forall$
"universal"	"existential"		"not exists"	"not for all"

called **bound** if quantified, otherwise **free**.

> Safety: possible to write expressions that generates *infinite* relations!

- *Example*: $\{ t \mid \neg\text{Employee}(t) \}$; all those tuples that are not employees!
  $\implies$ safe queries must return:

1. only values from the **active domain** (*values actually appearing in the database*!),
2. must be **finite**

- *Example*: $\{ t \mid \text{Person}(t) \wedge \neg\text{Employee}(t) \}$; all people who are not employees!

Domain relational calculus

- Now expresses results as *ranges over single values* from *domains of attributes*!

$$\{ x_1, \ldots, x_n \mid \text{Relation}(x_1, \ldots, x_n) \wedge (\text{ Conditions}(x_1, \ldots, x_n)) \}$$

where $x_1, \ldots, x_n$ are attributes of the relation.

Assertions

General integrity constraints are also available in *TRC* and *DRC*.
$\Rightarrow$ *expressed as predicates that must always be fulfilled*!
Example:

- Algebra: $\sigma_{\text{Price}<0}(\text{Product}) = \emptyset$
- TRC: $\neg\exists f(\text{Product}(f) \wedge f.Price < 0)$
- DRC: $\neg\exists I, N, P, C(\text{Product}(I, N, P, C) \wedge P < 0)$

Query evaluation

> Overview of **query processing**:

$$\text{Scanning, parsing, analysis} \Rightarrow \underbrace{\text{Query optimization}}_{\text{in relational algebra!}} \Rightarrow \text{Query code generator} \Rightarrow \text{Runtime database processor}$$

> Optimization in relational algebra is done by *decomposing the query into blocks*!
> **Query block**: a single `SELECT-FROM-WHERE` expression (possibly with `GROUP BY` and `HAVING`).
> $\Rightarrow$ *each block is then optimized separately*!

- which data is needed?
- where is the data, and how do we retrieve it (e.g. index)?
- how large is the expected result?

> Pipelining: Do not wait for 1 operation to finish, but instead *pass the results* of a previous operator *to the next operator* - *without waiting* to complete the previous operation!

For this, we need to consider the type of operations.
Types of selection queries:

1. *Point query*; Condition on a single value
   $\rightarrow$ very high selectivity!
2. *Range range*; Condition is on a range of values
   $\rightarrow$ selectivity depends
3. *Conjunction*; Combines logically two conditions with `AND`
   $\rightarrow$ selectivity often high
4. *Disjunction*; Combines logically two conditions with `OR`
   $\rightarrow$ selectivity often low

> Condition selectivity: Determined via. catalog information

$$\text{Selectivity} = \frac{\text{Tuples satisfying condition}}{\text{Total number of tuples in the relation}}$$

Projections are generally straightforwards, but *removing duplicates can require sorting/hashing*!

Aggregates can require different strategies depending on the operator:

- `MIN`, `MAX`: *full table scan* or *index-based search (!)*
- `SUM`, `COUNT`, `AVG`: *dense indexes (!)* or *full table scan* (materialized views can be good?)

Executing selection for *point query*:

- *Linear search* (brute-force!)
- *Binary search*, if: "ordered"
- *Index-based search*, if: "key attribute"

Executing selection for *conjunction/disjunction*:

- Start with most selective condition!

> $\Longrightarrow$ `JOIN` is by far the most *costly* operation!
> *NB*: strategies work on a *block* basis.

Overview of *join* strategies:

- *Nested loop join* (brute-force!)
- *Index-based join*
- *Sort-merge join*

- *Hash join*
  (Explained below...)

> Semi-join: Returns *only (!)* rows from *one relation* that has *at least* 1 match with the *other relation*.

- *Example*: $R \ltimes_A S = X$ with $R(A, B)$ and $S(A, C, D, \ldots)$ yields $X(A, B)$
- Reduces the number of tuples, which is beneficial in especially *distributed systems* where you are concerned *how much data is transferred*!
- Useful for *unnesting* `EXISTS`, `IN`, `ANY` subqueries because it only cares about 1 match.

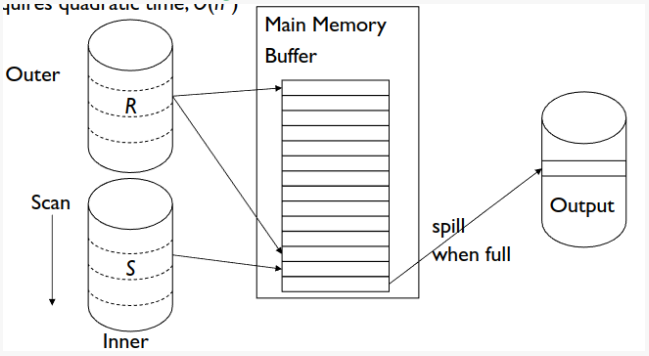
> Anti-join: Returns *only (!)* rows from the *first relation* where it *can not find a match* in *the second relation*.

- Useful for *unnesting negation subqueries*; `NOT EXISTS`, `NOT IN`, `ALL` because it rejects all matching rows.

> Join selection factor: Ratio of tuples in one relation that is expected to be joined with tuples in the other relation.

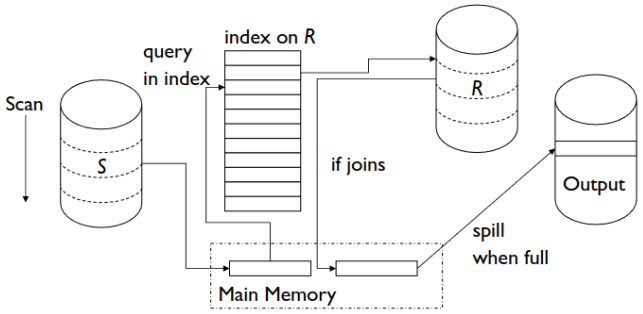
- *Generally*, for optimization we want tuples of the *outer relation* to match many tuples of the *inner relation* because we load outer relation blocks less often.

Nested loop join: $\mathcal{O}(n^2)$

> *NB*: Size of the *inner relation* should be *largest*!
>
> 
>

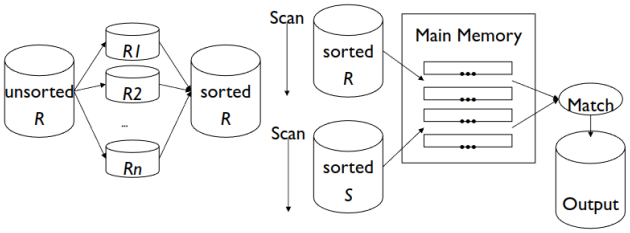
- Number of buffers in main memory determines how many blocks of the inner relation we can process simultaneously.
- Relation size of the outer dominates the cost.
- Can be be used to implement `LEFT/RIGHT/FULL OUTER JOIN`;
  $\rightarrow$ if `LEFT` use the *left relation* as the *outer relation* and *pad with nulls*!

Index-based join:



Sort-merge join:

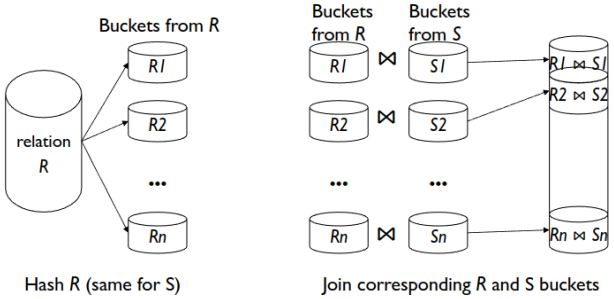
- *Most efficient* join is the **merge join**. To efficiently do this, we first *sort* the relations (or store them stored!) and *loads pairs of blocks* into memory and scans them!



> External sorting: Sorting algorithm *suitable for large files on disk that do not fit entirely in main memory*, such as most database files.

- We sort large files that do not fit in main memory, typically using a sort-merge strategy where subfiles, runs, of the main file are sorted and these runs are then merged.

Hash join:



Query optimization

Goal: Transform query into faster, equivalent query.

Two types of optimization:

1. *Heuristic (logical) optimization*
   ⇒ query tree (relational algebra)!
2. *Cost-based (physical) optimization*

Comparison:
Heuristic approach is more efficient to generate, but may not yield the optimal query plan.
Cost-based approach relies heavily on statistics gathered.

Heuristic optimization

- Translate into query tree of relational algebra and optimize.

Translate SQL query directly into tree:

1. leaves as input tables
2. apply operations sequential and build up

Steps to optimize:

1. Apply *selections* and *projections* early (consider *which are most restrictive*!).
2. Break up *conjunctive selections* into sequence of σ.
3. *Transform* $\sigma_C(R \times S) \Longrightarrow R \bowtie_C S$

Cost-based optimization

- Estimate and compare costs of queries with *different* strategies and choose the one *with the lowest estimate*.

- Most common strategy: *Bottom-up on the query tree, choose the best algorithm for implementing each operation*.

(!) Need to define a loss-function.

Approach:

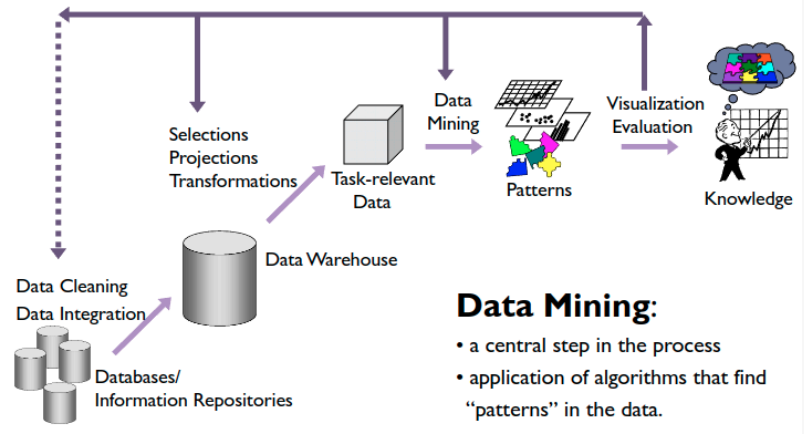
- Parameterize statistics on input relations
- Compare algorithm choices for operators
- Consider CPU-time, I/O-time, communication time, main-memory usage, or combinations.

Statistics: Stored in *system catalog*.

- number of records (tuples), r
- record size, R
- number of blocks, b
- blocking factor; how many records per block, bfr
- selectivity of an attribute, sl
- selection cardinality of an attribute, $s = sl \cdot r$
- join cardinality (estimated number of tuples from join)
  $\Rightarrow$ most important value for *joins*.

Data mining

- *Discovery of new information in terms of patterns or rules from vast amounts of data*.

> Knowledge-discovery in databases (*KDD*):
>
> 
>

> Association rules: Correlations between items in large datasets that appear to be associated with each other.

- *Example*: What items are frequently bought together?

$$\{\text{milk, butter, flour}\} \overset{\text{yields association}}{\Longrightarrow} [\text{milk} \Rightarrow \text{butter}]$$

1. ***Support***: percentage of transactions that contains all items in the rule (*prevalence*)
2. ***Confidence***: ratio between the transactions containing LHS, and how many of these transactions that contain the RHS (*strength*)

> **Apriori principle** (*monotonicity*):

- Any subset of a frequent itemset but also be frequent
  ⇒ *Any set that is not frequent can not be the subset of a frequent one*! (downward closure/anti-monotinicity)
  Approach

1. Split into 1-itemsets, and calculate thresholds.
2. Scratch any below the thresholds.
3. Perform joins on all (remaining) combinations of 1-itemsets to create 2-itemsets.
4. Repeat

> **Classification**: Learning a model able to describe different classes of data.

- *Supervised* as the classes to be learned are predetermined

> **K-means clustering**:

- (Initialization; partition objects into k sets.)
- Compute centroids of each cluster: $\mu_C = \frac{1}{|C|} \sum_{x_i \in C} x_i$
- Assign objects to nearest centroid.
- Repeat.

> Total-distance measure: $TD = \sqrt{\sum_{j=1}^{k} TD^2(C_j)}$

- for "good" clusters we generally want low TD.

> **Decision-tree classifiers**: Flow-chart tree performing decision based on class predictions.

- Nodes are *attributes*, branches are *outcomes*, and leaves are *predictors*.
  Building: created *top-down*!

1. Calculate total entropy based on predictors (YES/NO typically): $E_{tot} = -\sum_{i=1}^{n_{pred}} p_i \cdot \log_2(p_i)$
2. Calculate entropy and information gain for each attribute (consider only data for that attribute!):

$$E(A) = -\sum_{i=1}^{n_{\text{pred}}} p_i \cdot \log_2(p_i)$$

$$G(A) = E_{tot} - \sum_{i=1}^{n_{\text{outcomes}}} \frac{|O_i|}{|O|} \cdot E(T)$$

3. Use the attribute with *highest* information gain.
4. Repeat (now considering the data subsection)

Data warehouses

Traditional databases are optimized for querying and updating; **OLTP**; *"Online **Transactional** Processing"*. However, data warehouses are different:

> **OLAP**: "Online Analytical Processing", supports analysis of complex data (mostly *read-access*! - selectivity depends...)

> **DSS**: "Decision Support Systems", otherwise known as *EIS*; "Executive Information Systems" or *MIS*; "Management Information Systems"
> ⇒ supports making organization decisions based on historical data.

Typically;

1. *multiple databases*
2. *recurrent and predictable analysis*
3. *software designed specifically for requirements*

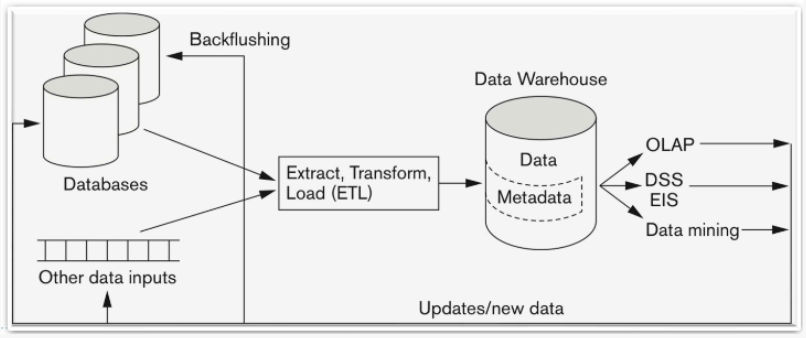
Views vs. Warehouse: Alike because both have read-only extracts, but

- Warehouses are *persistent storage*, while views are *materialized* on demand!
- and the structure is not at all alike (relational vs. multidimensional)!

Best database (OLTP) compromise is a *normalized* and *distributed* (scalability!) database.

Comparison with traditional databases:

- Warehouses optimized for data access, whilst databases for transactional and integrity measures.
- Warehouses emphasize historical data
- Warehouses are nonvolatile (not changed or deleted!)

> Structure:
>
> 
>
>
> ⇒
>
> 1. Extract, transform, load (*ETL*): cleaning and reformatting data fetched from databases
> 2. Data warehouse: sends data to *OLAP* or *data mining*

> **ETL** (*Extract, transform, load*):
>
> - Process of inserting data from transactional database(s).
> ⋆ different source databases with different schemas
> - Cleaning, validity, and quality of data.
> ⋆ incomplete data is corrected and backflushed to database
> - Convert data to model of data warehouse
> **(!)** Loading so much data is challenging, so typically it goes offline for regular intervals!

> Data modelling: Typically, *measures* are related to *several attributes* in a N-dimensional grid. Reasons for this is:
>
> 1. Better query performance
> 2. Non-volatile
> 3. High predictability based on expected analysis
> - *Example*: sales per. region/product/time period

⟹

> Multi-dimensional schemas: Data organized in **hyper cubes**.
>
> - *Dimension table*: Tuples of attributes of the dimension (typically denormalized!)
> - *Fact table*: Each tuple is a recorded fact, containing a measure or observed attribute(s). Identified with pointer(s) to dimension table!

- How to do SQL queries in such structure? ⇒ aggregates and `GROUP BY`.

> Star schema: Fact table with a single table for each dimension (creates star-pattern!).

> Snowflake schema: Dimensional tables from a star schema are organized into a hierarchy by normalizing them (creating branches and giving a snowflake-pattern!)

> Galaxy schema (*fact constellation*): set of tables that share some dimension tables (⇒ multiple fact tables).
>
> - Essentially, a collection of star schemas.
> - Hard to manage and support but very flexible.

> Navigating a warehouse by following *functionalities*;
>
> 1. *Roll-up*: move up in hierarchy (along 1 dimensional axis), so aggregate from smaller to larger regions (fine → coarse).
> 2. *Drill-down*: opposite of roll-up, so coarse → fine
> 3. *Pivoting*: rotate the reading axis
> - *slice* and *dice* (projections), *sorting*, *selection*, *derived attributes*

Major challenges: Quality control, complexity, evolution of source databases, change in usage.

Distributed database systems

> Distributed computing system:
>
> - Number of *processing sites* / *nodes*.
> - Interconnected by *computer network*.
> - *Cooperate* in performing tasks.
> - Achieves *more computing power*, but with overhead of *coordination*.

> Distributed database (**DDB**):
>
> - Process unit of execution (*transaction*!) in *distributed manner*.
> - Collection of *multiple logically related databases*.
> - Distributed over *computer network*.
> - DDBMS; "*Distributed Database Management System*", manages transparency to user

Transparency: *Hide implementation details* from end users. Users do not worry about operational details of network!

- *Location transparency*: access from any location
- *Naming transparency*: access to any named object from any site (requires unambiguous names!)
  ...
- *Fragmentation transparency*: allows storing tuples or attributes at different sites
- *Replication transparency*: allows storing copies of data at multiple sites
  $\Rightarrow$ *improves* reliability *(system running efficiently) and* availability *(system continuously available)*!

Query and update decomposition: A query or update must be decomposed into subqueries that can be executed at the individual sites.

Horizontal fragmentation (*sharding*): Horizontal subset of relation with tuples satisfying selection condition: $\sigma_C(R)$.
Derived horizontal fragmentation: Partitioning of primary relation to secondary relation via foreign keys. Meaning related data is fragmented in the same way.
Complete horizontal fragmentation: Every tuple in R satisfies $C_1 \text{ OR } \dots C_n$.
Disjoint complete horizontal fragmentation: No tuple in R satisfies $(C_i \text{ AND } C_j),\ i \neq j$.
Reconstruct R using `UNION`.

Vertical fragmentation: Vertical subset of relation as a subset of columns, $\Pi_{L_i}(R)$.
(!): *Each fragment must contain the primary key of the relation*!
Reconstruct R using `FULL OUTER JOIN`.
Complete vertical fragmentation: Set of vertical fragments include all attributes in R and share only primary key attributes.

$$L_1 \cup L_2 \cup \dots \cup L_n = \text{ATTRS}(R)$$
$$L_i \cap L_j = \text{PK}(R), \forall (i,j)$$

Mixed/hybrid fragmentation: Combination of vertical and horizontal fragmentation; $\Pi_{L_i}(\sigma_{C_i}(R))$.

Fragmentation *schema*: Definition of a *set* of fragments. Includes all tuples and attributes in database.
Allocation *schema*: Describes *distribution* of fragments to sites of distributed databases; *fully*, *partially replicated*, or *partitioned*.

Homogeneous distributed database system: All sites of database system have *identical* setup (*same database software system*!), e.g. Oracle, DB2, Sybase, etc.
(!) Underlying OS may be different: Linux, Windows, Unix, etc.

Heterogeneous distributed database system (*Federated*): Each site may run *different database system*, but data access is *a single* schema.
$\Rightarrow$ Each site must adhere to centralized access policy.
Issues:

1. Differences in data model (relational, object oriented, hierarchical, network, etc.)
2. Differences in constraints
3. Differences in query language

Catalog management

> ***Catalog***: databases with *metadata* about distributed database system

> Centralized catalogs: Entire catalog is stored in single site.
> ⇒ Easy to implement, but overload and bottleneck possible and also little reliability.

> Fully replicated catalogs: Each site stores copy of entire catalog. Updates are broadcasted.
> ⇒ Faster reads locally, but bottleneck for write-intensive loads.

> Partially replicated catalogs: Each site stores catalog information on data stored at the site. System tracks catalog entries from original site and sites with copies.
> ⇒ Updating copies may be delayed until access to data occurs.

Query processing

- ***Optimization criterion***: *minimize data transfer across network*!
- *e.g.* use **semi-joins** and in general consider data transfer!

Approach:
The input query is translated into an algebraic query on global relations, then mapped to separate queries on fragments, then optimized globally (typically with respect to communication cost), and finally optimized locally.

> MapReduce: *Efficient* and *scalable* distributed processing *automatically parallelized* with *runtime system*.
>
> - *Easy to distribute chunks, balance workload, handle node failure*.
> 1. Split data into chunks handled in *distributed manner*.
> 2. Map each record into key-value pairs in a list; $\text{"a"} \rightarrow \{\text{"a"}, 1\}$
> 3. Sort the list based on the key value.
> 4. Reduce key-value pairs by merging to a list of all values.
> - *Example*: Inverted-index (counting word occurrences)
> - *Example*: Sort-merge join

Concurrency control and recovery

- **main global consistency **and ==**recover all copies with consistency**==

> Primary site technique: Single site serves as coordinator for transaction management. Concurrency control and commits managed by primary site.
>
> - In *2PL* this site *manages locks*.
> **Disadvantages**: *overload primary site* and *primary site failure* (⇒ entire system will be inaccessible!)
> **Recovery**: Designated backup site (*shadow* of primary site).

> Primary copy technique: A data item partition is designed a *primary copy*. Simply lock the primary copy of the data item.
> **Advantage**: *no overload of site*
> **Disadvantage**: *identification of primary copy complex*, *distributed directory must be maintained*

> Voting: Send lock requests to sites with the data item, and if majority of sites grants lock then the transactions gets the data item

⟹

> **2PC** (*2-phase commit*): Transaction commits only when all nodes agree.
> If anything fails, then abort. Each node has it's own recovery scheme.
> *Global recovery manager* maintains information.
> **Phase 1**: All (participating) databases signal coordinator their part of the transaction has ended.
> **Phase 2**: If all votes (messages back to coordinator) is positive, then coordinator sends commit message. If not, then all participants undo locally.
> ⇒ If failure *after* "ready-to-commit" but *before* receiving "global-commit", the local site will be ***blocked*** (*does not know what to do*)!

⟹

> **3PC** (*3-phase commit*): Add "pre-commit" message to 2PC *between* voting and global decision.
> Coordinator receives all pre-commit messages *and then* sends global decision.
> ⇒ Means that participants receiving such a message know that all participants have voted commit!
>
> - *No blocking*!

Misc

> Parallel database: Builds on multiprocessor architecture, which can be either
>
> 1. *tightly coupled* with *shared primary memory* and *disk storage*,
> 2. *loosely coupled* with *secondary disk storage*, but *separate primary memory*.

There are 2 main ways of scaling a *database system*;
adding more resources or adding a new system.

> Horizontal scaling: When new systems (server racks!) are added to the existing system. This involves **sharding** (splitting data across servers) and hence *distributing* the database system.

> Vertical scaling: When new resources (CPU, RAM, etc.) are added to the existing system. This is easier to implement, and simply involves upgrading the current machine!

SQL is (*typically!*) **vertical scaling** because it does not easily *distribute across several nodes*. However, modern SQL databases are capable of horizontal scaling!

NoSQL is (*typically!*) **horizontal scaling** because it is designed to run on *several server nodes*. However, naturally they are also vertical scaling!