Algorithms

Mittwoch, 3. Mai 2017 13:17

Algorithms for Rel. Algebra

- Table Access
  - scan (load each page at a time)
  - index scan (if index available)
- Sorting
  - Two-phase external sorting
- Joins
  - (Block) nested-loops
  - Index nested-loops
  - Sort-Merge
  - Hashing (many variants)
- Group-by (~ self-join)
  - Sorting
  - Hashing

The **sort-merge join** (also known as merge join) is a join algorithm and is used in the implementation of a relational database management system.

The basic problem of a join algorithm is to find, for each distinct value of the join attribute, the set of tuples in each relation which display that value. The key idea of the sort-merge algorithm is to first sort the relations by the join attribute, so that interleaved linear scans will encounter these sets at the same time.

sort by key and then merge.
|R|log(|R|)+|S|log(|S|)+|S|+|R|
If it already is sorted or if the join key and the requested sorting are the same, then this method is used.
Also good for large datasets.

External sorting is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a hard disk drive. External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted subfiles are combined into a single larger file.

External merge sort [edit source]

One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together.[1][2] For example, for sorting 900 megabytes of data using only 100 megabytes of RAM:

1. Read 100 MB of the data in main memory and sort by some conventional method, like quicksort.
2. Write the sorted data to disk.
3. Repeat steps 1 and 2 until all of the data is in sorted 100 MB chunks (there are 900MB / 100MB = 9 chunks), which now need to be merged into one single output file.
4. Read the first 10 MB (= 100MB / (9 chunks + 1)) of each sorted chunk into input buffers in main memory and allocate the remaining 10 MB for an output buffer. (In practice, it might provide better performance to make the output buffer larger and the input buffers slightly smaller.)
5. Perform a 9-way merge and store the result in the output buffer. Whenever the output buffer fills, write it to the final sorted file and empty it. Whenever any of the 9 input buffers empties, fill it with the next 10 MB of its associated 100 MB sorted chunk until no more data from the chunk is available. This is the key step that makes external merge sort work externally -- because the merge algorithm only makes one pass sequentially through each of the chunks, each chunk does not have to be loaded completely; rather, sequential parts of the chunk can be loaded as needed.

https://en.wikipedia.org/wiki/External_sorting

Nested Loop Join

Two relations R and S are joined as follows:

```
For each tuple r in R do
   For each tuple s in S do
      If r and s satisfy the join condition
         Then output the tuple <r,s>
```

works really well if R and S are small, and otherwise it's not so nice. Complexity |R|× |S| can explode fast.

Block Nested Loops

A **block-nested loop** (**BNL**) is an algorithm used to join two relations in a relational database.[1]

This algorithm[2] is a variation on the simple nested loop join used to join two relations R and S (the "outer" and "inner" join operands, respectively). Suppose $|R| < |S|$. In a traditional nested loop join, S will be scanned once for every tuple of R. If there are many qualifying R tuples, and particularly if there is no applicable index for the join key on S, this operation will be very expensive.

The block nested loop join algorithm improves on the simple nested loop join by only scanning S once for every *group* of R tuples. For example, one variant of the block nested loop join reads an entire page of R tuples into memory and loads them into a hash table. It then scans S, and probes the hash table to find S tuples that match any of the tuples in the current page of R. This reduces the number of scans of S that are necessary.

A more aggressive variant of this algorithm loads as many pages of R as can be fit in the available memory, loading all such tuples into a hash table, and then repeatedly scans S. This further reduces the number of scans of S that are necessary. In fact, this algorithm is essentially a special-case of the classic hash join algorithm.[*citation needed*]

The block nested loop runs in $O(P_r P_s / M)$ I/Os where M is the number of available pages of internal memory and P_r and P_s is size of R and S respectively in pages. Note that block nested loop runs in $O(P_r + P_s)$ I/Os if R fits in the available internal memory.

Hash Join

Classic hash join [edit source]

The classic hash join algorithm for an inner join of two relations proceeds as follows:

- First prepare a hash table of the smaller relation. The hash table entries consist of the join attribute and its row. Because the hash table is accessed by applying a hash function to the join attribute, it will be much quicker to find a given join attribute's rows by using this table than by scanning the original relation.
- Once the hash table is built, scan the larger relation and find the relevant rows from the smaller relation by looking in the hash table.

The first phase is usually called the **"build" phase**, while the second is called the **"probe" phase**. Similarly, the join relation on which the hash table is built is called the "build" input, whereas the other input is called the "probe" input. It is like merge join algorithm[*clarification needed*].

This algorithm is simple, but it requires that the smaller join relation fits into memory, which is sometimes not the case. A simple approach to handling this situation proceeds as follows:

1. For each tuple r in the build input R
    1. Add r to the in-memory hash table
    2. If the size of the hash table equals the maximum in-memory size:
        1. Scan the probe input S, and add matching join tuples to the output relation
        2. Reset the hash table, and continue scanning the build input R
2. Do a final scan of the probe input S and add the resulting join tuples to the output relation

This is essentially the same as the block nested loop join algorithm. This algorithm scans S more times than necessary.

https://en.wikipedia.org/wiki/Hash_join

only works for equijoins. Idea is that the hashfunction creates buckets. (Maybe hash the bucket again if they're many.)

Idea is that one bucket will always fit in main memory.
O(|R| + |S|)

If RAM bucket is full, just write it to hdd, when it is full again, append on HDD. Idea is that this whole hdd block will in the end still fit into main memory. So we need roughly relationsize/n main memory to load back from hdd and n from just the buckets in RAM.
estimate: hashjoin needs at least the square root of relation size.
Goal: to minimize (n, R/n) by choosing n. This is minimal for n approx sqrt(R)

Relational Algorithms

Donnerstag, 8. Juni 2017 13:56

Algorithms
When to use which one

sort-merge join
sorts the relations by the join attribute, then runs interleaved scans to merge the rows.
$R \log(R) + S \log(S) + S + R$
Useful when the data is already sorted, and for large datasets

External merge-sort
1. Read RAM-size in main memory and sort
2. write sorted data to disk
3. repeat until all data is in sorted RAM-sized chunks
4. read the first few MB of each chunk so that they fit into ram
5. merge those into an output buffer and when this is full, write it into a final array. When a input buffer empties, refill.

Nested Loop Join
Simply a double loop. if r and s satisfy the join condition, then join.
works well if R and S are small, otherwise it's bad
$R * S$ can explode fast.

Block Nested Loops
suppose R < S. Only scan S once for every group of R tuples - i.e. read an entire page of R tuples into memory, load them into a hash table. Then scan S and probe the hash table to find matching tuples in the hash table.
$\frac{P_r P_S}{M}$ where M is the number of available pages of internal memory and the P are the sizes of R and S in pages.
This runs in $O(P_r + P_s)$ if R fits in the available internal memory.

Hash Join
Prepare a hash table and then scan the larger relation and find the relevant rows from the smaller relation within the hash table.
This requires that the smaller join relation fits into memory and only works for equijoins. Useful if we often join on the same attribute of the smaller relation because then we can reuse the hash table.
Idea is that one bucket of the hash function will always fit in main memory.
$O(R + S)$

If RAM bucket is full, write it to hdd. When it is full again, append it on hdd. So we need roughly relationsize/n main memory to load back from hdd and n for those that are already in RAM.
Estimate: hashjoin needs at least $\sqrt{Relationsize}$ because we need to minimize $\left(n, \frac{R}{n}\right)$ by choosing the amount of Buckets n. This is minimal for $n \approx \sqrt{R}$

* $max(k, N_p/k)$ results from two phases of GHJ: partitioning and build&probe. During partitioning we need at least k (#partitions) pages to accommodate all partitions. During build&probe, we need N_p/k (N_p: #pages for the smaller relation) pages on average to accommodate the hash tables. Thus, the maximum of these two numbers dictates how much buffer we need. Now, we can select a k', in order to minimize the needed buffer space:

$$k' = \underset{k}{\operatorname{argmin}}\, max(k, N_p/k) = \sqrt{N_p}$$

ACID

Donnerstag, 8. Juni 2017 13:06

Atomicity: a transaction is executed in entirety or not at all
Consistency: a transaction executed completely on a consistent database yields a consistent result
Isolation: a transaction executes as if it were alone in the system
Durability: commited changes of a transaction are never lost - can be recovered.

2PC vs 3PC

Donnerstag, 8. Juni 2017 07:17

2PC
Coordinator sends VOTE-REQ to all
Participants recieve that and vote YES or NO
Coordinator waits for all participants until first NO
    all YES => commit and sends COMMIT
    some NO => abort and sends ABORT to all which voted YES
        Those who voted NO have already aborted themselves
Participant recieves COMMIT or ABORT and does that, then stops

This Protocol meets the 5 AC rules:
AC1: Every processor decides the same
AC2: Any processor arrving at a decision stops => Cannot reverse its decision
AC3: Controller only decides COMMIT if nobody voted NO => No imposed COMMIT
AC4: If there are no failures and all processors voted YES, the decision will be COMMIT (nontriviality)
AC5: If all failures are repaired and no more failures occur for sufficiently long, then all processors will eventually reach a decision (liveness)

For AC5 we need to extend the protocol and ask around in case of timeout.

Uncertainity Period: When a participant times out waiting for a decision and everybody is in the same situation when asking around, all processors will block. This can happen if the coordinator fails after receiving all YES votes but before sending any COMMIT message
Why can't every participant then just ask everybody else? If one says no, abort, else say yes. Because the failed coordinator might want to abort.
There's also the possibility that the coordinator and a participant fail. In that case, it is impossible to say whether this participant has recieved the COMMIT and committed or whether we should abort because no COMMIT message was sent, so we have to wait.

Persistence through logging to node disk.
YES logs *before* sending, NO logs before or after. Because if it crashes in between and finds neither a YES nor a NO log record, it aborts unilaterally.
Same for the coordinator with COMMIT or ABORT.
Reason is probably that data to evaluate is then no longer in memory and cannot be reevaluated if not yet decided.
https://courses.cs.washington.edu/courses/csep552/13sp/lectures/4/2pc.pdf

Linear 2PC
Less messages by moving on in a daisychain. Total number of messages is not 3n but only 2n because a NO propagates in both directions and a COMMIT through the whole line. The coordinator seems to be the end of the chain.

3PC
Doesn't block => liveness
AC1: every node decides the same
AC2: no node changes its decision
AC3: no imposed COMMIT
AC4: nontriviality: if there are no failures and everybody voted YES, then the decision will be COMMIT
AC5: If all failures are repaired and there aren't any more for sufficiently long, then the protocol will terminate with a decision (liveness)

Assuming no communication failures.
NB rule: Nobody can decide to commit as long as anybody is uncertain.

Difference to 2PC: PRE-COMMIT -> ACK -> COMMIT
So if the coordinator fails after VOTE-REQ and all processors vote YES, then they would all be in uncertainity in 2PC. In 3PC it is guaranteed that nobody has decided to commit while anybody is uncertain. So if everybody is uncertain, they can find that out and safely abort. To make sure that nobody is uncertain before deciding, the coordinator needs the PRE-COMMIT. Now if the coordinator crashes after sending PRE-COMMIT, participants know what is going to happen but have to ask around to make sure everybody is certain before committing.
If coordinator times out waiting for votes, ABORT. If coordinator times out waiting for ACKs, ignore those and send the others a commit. (**some also say to wait. The appended solutions say otherwise**) They can later ask around to find out that they should commit.
If a node fails after recieving PRE-COMMIT, it has to ask around to make sure nobody is uncertain.

Again, logging YES before sending because if crashes and no YES there, then abort. It seems like sending precommits is not logged, so if the coordinator crashes after starting 3PC but has no decision in its log, it has to ask around (maybe somebody already got a precommit)

Not used in practise because probability of blocking is small enough and 3PC is too expensive.

The following sequence of events shows an execution of the 3PC protocol where no failures occur:

time step	event
1	$(C, P_1, request)$
2	$(C, P_2, request)$
3	(P_1, C, yes)
4	(P_2, C, yes)
5	$(C, P_1, pre\text{-}commit)$
6	$(C, P_2, pre\text{-}commit)$
7	(P_1, C, ack)
8	(P_2, C, ack)
9	$(C, P_1, commit)$
10	$(C, P_2, commit)$

We now modify this sequence of events starting from some time step. Complete each new sequence with one possible next event such that it models a valid execution of the 3PC protocol.

Sequence (i):

time step	event
4	(P_2, C, no)
5	$\mathbf{(C, P_1, abort)}$

Sequence (ii):

time step	event
2	$(C, fail)$
3	(P_1, C, yes)
4	$\mathbf{(P_1, ask\ around \rightsquigarrow abort)}$

Sequence (iii):

time step	event
5	$(C, fail)$
6	$\mathbf{(P_1, ask\ around \rightsquigarrow abort)}$

Sequence (iv):

time step	event
6	$(C, fail)$
7	$\mathbf{(P_2, ask\ around \rightsquigarrow commit)}$

Sequence (v):

time step	event
4	$(P_2, fail)$
5	$\mathbf{(C, P_1, abort)}$

Sequence (vi):

time step	event
6	$(P_2, fail)$
7	$(C, P_2, pre\text{-}commit)$
8	(P_1, C, ack)
9	$\mathbf{(C, P_1, commit)}$

Termination Protocol

- Elect a new coordinator.
- New coordinator sends a “state req” to all processes. Participants send their state (aborted, committed, uncertain, committable).
- TR1 = If some “aborted” received, then abort.
- TR2 = If some “committed” received, then commit.
- TR3 = If all uncertain, then abort.
- TR4 = If some “committable” but no “committed” received, then send “PRE-COMMIT” to all, wait for ACKs and send commit message.

TR4 is similar to 3PC, have we actually solved the problem?

- Yes, failures of the participants on the termination protocol can be ignored. At this stage, the coordinator knows that everybody is uncertain, those who have not sent an ACK have failed and cannot have made a decision. Therefore, all remaining can safely decide to commit after going over the pre-commit and commit phases.
- The problem is when the new coordinator fails after asking for the state but before sending any pre-commit message. In this case we have to start all over again.

Normal Forms & MVD

Donnerstag, 8. Juni 2017 09:34

Sheet 9 - Minker NF, Normal Forms Slides, MVD, MVD Wikipedia

1NF: All **nonkey** attributes have to depend on the key. ⋆
2NF: the/a **whole** key (else, split the table to avoid redundancy)
3NF: and **only** the key (directly, no transitivity, else you take longer to access)
BCNF: The same holds for **key** attributes (only different if we have overlapping keys, or in a case like below)

Aso de merksatz isch eifach ambiguous, aber ich merks mir demfall eifach mit "All nonkey attributes must depend on the key (1NF), the whole key (2NF) mit transitivität erlaubt, and only the superkey (3NF) mit transitivität nöd erlaubt aber halt direkti supersets scho)"

> In addition to the primary key, the relation may contain other candidate keys; it is necessary to establish that no non-prime attributes have part-key dependencies on **any** of these candidate keys.
> Aus <https://en.wikipedia.org/wiki/Second_normal_form>

⋆ 1NF also includes other basic stuff like
An attribute can only have one value
A row has to be unique
A row has to be determined by a key uniquely

Formally:
2NF: For every functional dependency X -> Y, one of the following holds
Y is part of X (trivial dependency)
Y is part of a key (because not nonkey)
X is a key or a superset of a key (depends on a whole key or more)
There is no attribute in X that belongs in a key (=> does not depend only partly on key)

3NF: ~~There is no attribute in X that belongs in a key~~
This statement suffices no more, we got rid of some transitivity. The other three still hold:
Either
Y is part of X
Y is part of a key (because not nonkey)
X is a key or a superset of a key

BCNF: ~~Y is part of a key (because not nonkey)~~
It now has to hold also for key attributes. What remains is
Y is part of X
X is a key or a superset of a key

4NF: if the relation is in BCNF
AND for every non-trivial **multivalued dependency** X ->> Y, X is a superkey. That is, X is either a candidate key (minimal) or a superset of a key.

> A multivalued dependency exists when there are at least 3 attributes (like X,Y and Z) in a relation and for value of X there is a well defined set of values of Y and a well defined set of values of Z. However, the set of values of Y is independent of set Z and vice versa.

X ->> Y falls Y und Z sich nicht implizieren aber als Mengen klar definiert sind für jedes X.
Trivial MVD ist es wenn Y subset von X oder X ∪ Y die ganze Relation ist.

- $\alpha \rightarrow\rightarrow \beta$ iff
  - $\forall$ t1, t2 $\in$ R: t1.α = t2.α $\Rightarrow \exists$ t3, t4 $\in$ R:
    - t3.α = t4.α = t1.α = t2.α
    - t3.β = t1.β, t4.β = t2.β
    - t3.γ = t2.γ , t4.γ = t1.γ

"MVD ist es genau dann, wenn für alle Reihenpaare (t1,t2) ein Reihenpaar existiert, sodass alle kombinationen von β und γ abgedeckt sind, die beiden also einander nicht implizieren, und alle Reihen das selbe α haben."
Also geht es darum, ob es mehrere kombinationen gibt, die eigentlich unabhängig von einem anderen key sind.

BCNF vs 3NF
R{A,B,C} where {A,B} is a key. Given the dependency C->B, R satisfies the requirements of 3NF but not BCNF.

Aus <https://stackoverflow.com/questions/8437957/difference-between-3nf-and-bcnf-in-simple-terms-must-be-able-to-explain-to-an-8>

- Result
  - any schema can be decomposed losslessly into BCNF
  - but, preservation of dependencies cannot be guaranteed
  - need to trade „correctness“ for „efficiency“
  - that is why 3NF is so important in practice

See also https://stackoverflow.com/a/33379413/2550406

The 3NF problem: The partial key/prime attribute "Court" is dependent on something other than a superkey. Instead, it is dependent on the partial key/prime attribute "Rate Type". This means that the user must manually change the rate type if we upgrade a court, or manually change the court if wanting to apply a rate change.

- But what if the user upgrades the court but does not remember to increase the rate? Or what if the wrong rate type is applied to a court?

(In technical terms, we cannot guarantee that the "Rate Type" -> "Court" functional dependency will not be violated.)

The BCNF solution: If we want to place the above table in BCNF we can decompose the given relation/table into the following two relations/tables (assuming we know that the rate type is dependent on only the court and membership status, which we could discover by asking the clients of our database, the owners of the tennis club):

FD->MVD
When X -> Y, then also X ->> Y

Lossless Decomp
the decomposition is lossless iff
$(R_1 \cap R_2) \rightarrow R_1$ *or* $(R_1 \cap R_2) \rightarrow R_2$
The formal definition also says that it must fulfill $R_1 \cup R_2 = R$, but this simply means that it's not ok if some element is lost or added - that should be obvious anyway.

Argumentation using FDs

5

decomposition of a relation R into R1 and R2 is Lossless join decomposition if you can construct back R by joining the relation R1 and R2(form R1 ⋈ R2 you can obtain R).

For a decomposition of Relation R into R1 and R2 to be lossless it must satisfy any of 2 condition:

```
1. R1 ∩ R2 -> R1
2. R1 ∩ R2 -> R2
```

If the above relation doesn't make any sense then think of it like this when you are intersecting 2 relation R1 and R2 and obtaining common attributes then if the common attributes are able to determine any one of the relation then this common attribute are candidate key for the obtained relation(think why ?) and hence you can join using this candidate key the other relation to obtain R.

Regarding dependency preserving a decomposition of relation R is dependency preserving if the Functional dependency of R can be obtained by taking the union of the functional dependency of all the decomposed relation.

share edit flag

answered Nov 8 '14 at 10:20

akashchandrakar
824 ●2 ●10 ●37

Dependency Preserving Decomp
Keeps the FDs within the same table

minimal Basis & Synthesis

Donnerstag, 8. Juni 2017 13:21

Minimal Basis

minimal basis

1. Split all FDs so that there is one FD for each right side

(

1.5 Test every left hand side and remove parts of it that are implied by the rest of the lefthand side.

// Use this to be on the safe side, so we can always ignore the FD that we're changing both in lefthand and righthand reductions

)

2. For each FD, try to remove each lefthand-element and deduce the right side with the other FDs, including the one we're trying to change. If it works, remove that element, else try the next.
3. remove redundant dependencies by trying to come from its left side to its right side without that FD itself
4. merge right sides back together for the same lefthand sides.

Synthesis Algorithm -> 3NF

1. Compute the Minimal Basis
2. for all FD X -> Y, … in the minimal basis, create a relation {X,Y, …}
3. create a relation for ~~one~~ all keys of the original relation
4. remove all relations that are subsets of other relations

Proof that this gives 3NF

Synthesis Algo -> 3NF only

- Let $\mathcal{R}_i$ be a relation created by the Synthesis Algo
- Case 1: $\mathcal{R}_i$ was created in Step 3 of the algo
  - $\mathcal{R}_i$ contains a key of $\mathcal{R}$
  - there are no non-trivial FDs in $\mathcal{R}_i$
  - $\mathcal{R}_i$ is in 3NF
- Case 2: $\mathcal{R}_i$ was created in Step 2 by an FD: $\alpha \rightarrow \beta$
  - (1) $\mathcal{R}_i := \alpha \cup \beta$
  - (2) α is a key of $\mathcal{R}_i$
    - α is minimal because of left reduction of minimal basis
    - $\alpha \rightarrow \mathcal{R}_i$ by construction of $\mathcal{R}_i$
  - (3) $\alpha \rightarrow \beta$ is not evil because α is a superkey of $\mathcal{R}_i$
  - (4) Let $\gamma \rightarrow \delta$ be any <u>other</u> non-trivial FD ($\gamma \rightarrow \delta = \alpha \rightarrow \beta$)
    - $\delta \subseteq \alpha$ because of right reduction in minimal basis and because $\alpha \rightarrow \gamma$
    - δ contains only attributes of a key; $\gamma \rightarrow \delta$ is not evil qed

2. Apply the synthesis algorithm.

{ShipType, ShipName}
{Cargo, ShipName, TripId}
{Date, ShipName, TripId, Port}
{ShipName, Date}

I only need to add a relation for one of the keys, right?

You need to add relations for all keys.

3NF->BCNF

1. Determine BCNF:
For relation R to be in BCNF, all the functional dependencies (FDs) that hold in R need to satisfy property that the determinants X are all superkeys of R. i.e. if X->Y holds in R, then X must be a superkey of R to be in BCNF.

In your case, it can be shown that the only candidate key (minimal superkey) is ACE. Thus both FDs: A->B and C->D are violating BCNF as both A and C are not superkeys or R.

2. Decompose R into BCNF form:
If R is not in BCNF, we decompose R into a set of relations S that are in BCNF.
This can be accomplished with a very simple algorithm:

```
Initialize S = {R}
While S has a relation R' that is not in BCNF do:
   Pick a FD: X->Y that holds in R' and violates BCNF
   Add the relation XY to S
   Update R' = R'-Y
Return S
```

In your case the iterative steps are as follows:

```
S = {ABCDE}       // Intialization S = {R}
S = {ACDE, AB}    // Pick FD: A->B which violates BCNF
S = {ACE, AB, CD} // Pick FD: C->D which violates BCNF
// Return S as all relations are in BCNF
```

Thus R(A,B,C,D,E) is decomposed into a set of relations: R1(A,C,E), R2(A,B) and R3(C,D) that satisfies BCNF.

Note also that in this case, functional dependency is preserved but normalization to BCNF does not guarantee this.

Lossless Decomposition

Recall that we learned how to "normalize" relations (i.e., put them in BCNF) by decomposing their schemas into two or more sets of attributes
Example: `Enroll(student, class, TA)`

- In any given class, each student is assigned to exactly one TA
- One TA can assist only one class

Recall that a relation R is in *BCNF* if for every nontrivial FD $X \rightarrow Y$ in R, X is a superkey

- $X \rightarrow Y$ is a *BCNF violation* if it is nontrivial and X does not contain any key of R
- Based on a BCNF violation $X \rightarrow Y$, decompose R into two relations:
  - One with $X \cup Y$ as its attributes (i.e., everything in the FD)
  - One with $X \cup (attrs(R) - X - Y)$ as its attributes (i.e., left side of FD plus everything not in the FD)

Example: turn `Enroll` into BCNF
- BCNF violation:
- Decomposed relations:

What does this decomposition "work"? Why can't we just tear sets of attributes apart as we like?

- $\rightsquigarrow$ The decomposed relations need to represent the same information as the original
- $\rightsquigarrow$ We must be able to reconstruct the original from the decomposed relations

Decomposition to 4NF

http://infolab.stanford.edu/~ullman/fcdb/spr99/lec14.pdf
Start in 3NF
For every MVD $X \rightarrow Y$ (evil), remove its relation from R and add instead
$R_1 = X \cup Y$
$R_2 = R - Y$

Integrity Constraints

Donnerstag, 8. Juni 2017 10:59

Integrity Constraints

unique for alternative keys
foreign key / **references** references columns within the same database but maybe another table. if an entry in one column has to be existent in a different column, then it is a foreign key / referenced key.
In InnoDB there must be an index where the foreign key columns are listed as the first columns in the same order (for speed). The keyword *foreign key* is written in the child table. The parent table column contains the valid values. If something in the parent is deleted, the ON DELETE in the child declaration triggers.

Maintaining:
cascade propagate the updates or deletes. So deletes all child rows that referenced this.
restrict prevent deletion of the primary key before attempting the change. Causes an error.
no action prevents modifications but might trigger something. Causes an error.
set default, set null sets references to null or default value when they are updated/deleted

ECA: Event, Condition, Action

Example Syntax:

```
create table table_S
        (…, k integer references table_R
                ON UPDATE CASCADE );
```

Constraints on Domains
check s **between** 1 **and** 13
check level **in** (`Assistant´, 'Associate', 'Full')
check (begin_date < end_date)

```
ALTER TABLE Persons ADD CHECK (age>=18);
```

Within a "CREATE TABLE" block:

```
age int CONSTRAINT CHK_PersonAge CHECK (age>=18);
```

or

```
age int check(age>=18)
```

```
ALTER TABLE Persons DROP CONSTRAINT CHK_PersonAge;
```

```
create table Assistant
    ( PersNr            integer primary key,
    Name                varchar(30) not null,
    Area                varchar(30),
    Boss                integer,
    foreign key (Boss) references Professor
                        on delete set null);
create table Lecture
    ( Nr        integer primary key,
    Title               varchar(30),
    CP                  integer,
    PersNr              integer references Professor
                        on delete set null);
```

Events
ON UPDATE, AFTER UPDATE, BEFORE UPDATE
=> we can use old.Level and new.Level, but with a colon before it in oracle. Not in mysql and postgresql tho.

Triggers
Instead of in the table itself

```
CREATE TRIGGER my_trigger
AFTER UPDATE OR DELETE
ON Persons
DECLARE
    my_action other_table.action
BEGIN
    IF UPDATING THEN
        my_action := 'update';
    ELSIF DELETING THEN
        my_action :='delete';
    END IF;

    INSERT INTO …. do something else
END
```

Query optimization

Donnerstag, 8. Juni 2017 14:49

Perform selection and projection early
Perform most restrictive selection and join operations before similar operations.
Some systems use heuristics.
=> TODO: Look at Query Tree

Notes Ex '16

Freitag, 9. Juni 2017 10:38

Like operator
WHERE column LIKE pattern
% stands for 0 or more characters
_ stands for a single character
regex supported with REGEXP instead of LIKE. In Postgresql, this is regexp_matches("reg{e|ex}p"}

Where equal
With a single equality sign. =, not ==
Not equal is <>
BETWEEEN 1 AND 3 is the same as <= 3 and >= 1

Trick Question: remember Candidate key must be minimal

DELETE FROM
has no asterisk

Default order of ORDER BY
ASC

Mysql JOIN ON vs USING

ON is the more general of the two. One can join tables ON a column, a set of columns and even a condition. For example:

```
SELECT * FROM world.City JOIN world.Country ON (City.CountryCode = Country.Code) WHERE ...
```

USING is useful when both tables share a column of the exact same name on which they join. In this case, one may say:

```
SELECT ... FROM film JOIN film_actor USING (film_id) WHERE ...
```

An additional nice treat is that one does not need to fully qualify the joining columns:

```
SELECT film.title, film_id # film_id is not prefixed
FROM film
JOIN film_actor USING (film_id)
WHERE ...
```

Group By
is on the selection, not before it.

Forign Key without any trigger is the same as RESTRICT

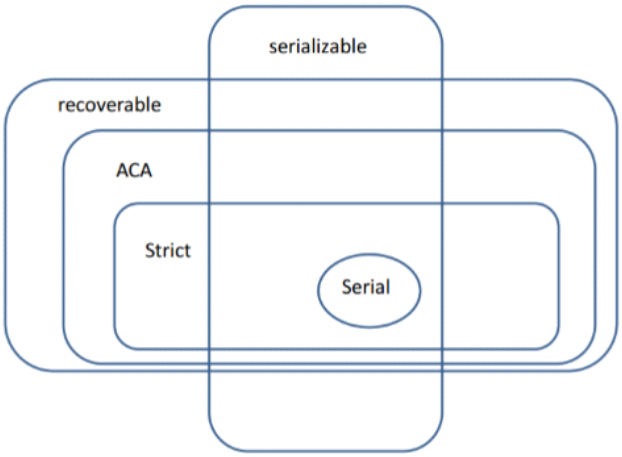
ER
usually, the relationkey does not contain the attributes of the relation. in an 1-N relation, one entity specified as key suffices.

is-a has the arrow pointing to the container. e.g. man -> person <- woman
is-a need not be complete: A Person can be neither man nor woman. Or both.

strict, aca, recoverable

Freitag, 9. Juni 2017 14:03

Every history can be ordered in two classes relative to recoverability and serializability:



If a Transaction T2 writes to a data location A, which T1 later reads, then a history is:

i Recoverable if $c_2 <_H c_1$

ii Avoiding Cascading Abort (ACA) if $c_2 <_H r_1(A)$

iii Strict if $c_2 <_H o_1(A)$ where $o_1(A)$ is a read or write

- A transaction T_1 reads from another transaction T_2 if T_1 reads a value written by T_2 at a time when T_2 was not aborted
- Recoverable (RC) history
  - If T_i reads from T_j and commits, then $c_j < c_i$
- Avoids cascading aborts (ACA) history
  - If T_i reads x from T_j, then $c_j < r_i[x]$
- Strict (ST) history
  - If T_i reads from or overwrites a value written by T_j Tj then $c_j < r_i[x]$/ $w_i[x]$ or $a_j < r_i[x]$/ $w_i[x]$

Aufpassen: wenn 1 zuerst liest und dann 2 schreibt, danach beide committen, dann ist es trotzdem strict.

What do they mean?

- Recoverable:
  - No need to undo a committed transaction
- ACA:
  - Aborting a transaction does not cause aborting others
- Strict:
  - Undoing a transaction does not undo the changes of other transactions

Given $w_2(A), w_1(A), r_2(A), w_2(A), c_1, c_2$, we can reform the history to $w_1(A), c_1, r_2(A), w_2(A), w_2(A), c_2$ and the result stays the same.
The read still gets the same value of A and the final value of A is the same as well. Is this a legal serialization? Why/Why not?
A: We aren't allowed to change the order within the same transaction.

For s2PL, 2PL, ss2PL see Sheet 11 - Minker

> To comply with **strong strict two-phase locking** (SS2PL) the locking protocol releases both *write (exclusive)* and *read (shared)* locks applied by a transaction only after the transaction has ended, i.e., only after both completing executing (being *ready*) and becoming either *committed* or *aborted*. This protocol also complies with the S2PL rules. .

Sheetback

Sonntag, 11. Juni 2017 11:47

NOT IN
SELECT * FROM employees ee WHERE
ee.emp_no NOT IN
(SELECT e.emp_no
FROM employees e
JOIN dept_emp de ON e.emp_no = de.emp_no)

// we could also do a MINUS instead of EXCEPT (but not in postgresql). They are synonymous in Oracle SQL.
// The difference (minus) returns repetitions in Oracle
The NOT IN works good in where clause, the EXCEPT works on the set level, like e.g. UNION
The except does not return repetitions. Unless writing "except all"

right outer join
⋈ [takes join plus all remaining right sides
⋈ takes join plus all remaining left sides

ORDER BY
sorts ascending by default

```
ORDER BY column1 DESC, column2
```

This sorts everything by `column1` (descending) first, and then by `column2` (ascending, which is the default) whenever the `column1` fields for two rows are equal.

this regards column2 only where column1 is the same

GROUP BY

`Group By X` means **put all those with the same value for X in the one group**.

`Group By X, Y` means **put all those with the same values for both X and Y in the one group**.

GROUP BY needs to include all non-aggregates. Not sure why. Maybe so it knows in which order to sort them internally. This is different in other implementations but Postgresql needs a primary key to group by, or if you don't select the primary key then a complete specification.

NULL values are together in one group

GROUP BY only prints the same grouped elements once. Useful for aggregate functions (splitting into groups. otherwise use order by)

INTERSECT, UNION ALL
INTERSECT takes distinct values, so does UNION
UNION ALL takes also duplicate rows, as does NATURAL JOIN

HAVING

`WHERE` clause introduces a condition on *individual rows*; `HAVING` clause introduces a condition on *aggregations*, i.e. results of selection where a single result, such as count, average, min, max, or sum, has been produced from *multiple* rows. Your query calls for a second kind of condition (i.e. a condition on an aggregation) hence `HAVING` works correctly.

As a rule of thumb, use `WHERE` before `GROUP BY` and `HAVING` after `GROUP BY`. It is a rather primitive rule, but it is useful in more than 90% of the cases.

… HAVING DATEPART(year, dep.from_date) > 1990

```
FROM employees e
HAVING COUNT(
      SELECT * FROM dept_emp de1, dept_emp de2
      WHERE de1.emp_no=e.emp_no
      AND de2.emp_no=emp_no
      AND de1.dept_no<>de2.dept_no
)>1
```

You cannot use COUNT with nested subqueries. What you could do is move it to the nested SELECT ("SELECT COUNT(*) FROM ..."). Then, you would need to use WHERE instead of HAVING to get the result you need. HAVING is used for checking conditions related to aggregated results when GROUP BY is used.

1. You can have both. These two do different things. WHERE compares single elements while HAVING compares results of aggregating functions (AVG, SUM, MAX, etc.) when the elements are grouped by something. Normally, the query "works" this way: First, WHERE is used to get valid single results. Then, the results are aggregated (using GROUP BY or simply aggregating all values into one). Lastly, HAVING is used to get only those results which fulfill some condition after the aggregation. For example:
HAVING SUM(salary) > 100

Converting to Float

+0.0

or

CAST (SUM(alpha) AS FLOAT)

Aggregates

SUBSTRING (expression ,start , length)

start

Is an integer or bigint expression that specifies where the returned characters start. (The numbering is 1 based, meaning that the first character in the expression is 1). If start is less than 1, the returned expression will begin at the first character that is specified in expression. In this case, the number of characters that are returned is the largest value of either the sum of start + length- 1 or 0. If start is greater than the number of characters in the value expression, a zero-length expression is returned.

SELECT COUNT(*) …

counts everything else in the selection

WITH

The basic value of SELECT in WITH is to break down complicated queries into simpler parts. An example is:

```
WITH regional_sales AS (
        SELECT region, SUM(amount) AS total_sales
        FROM orders
        GROUP BY region
     ), top_regions AS (
        SELECT region
        FROM regional_sales
        WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
     )
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

The optional RECURSIVE modifier changes WITH from a mere syntactic convenience into a feature that accomplishes things not otherwise possible in standard SQL. Using RECURSIVE, a WITH query can refer to its own output. A very simple example is this query to sum the integers from 1 through 100:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

The general form of a recursive WITH query is always a *non-recursive term*, then UNION (or UNION ALL), then a *recursive term*, where only the recursive term can contain a reference to the query's own output. Such a query is executed as follows:

Solution:

```
WITH RECURSIVE enames AS (
  SELECT first_name, last_name, emp_no, 0 AS len
  FROM employees
  UNION ALL
  SELECT p.first_name, p.last_name, e.emp_no, e.len+1 AS len
```

notes temp

Mittwoch, 7. Juni 2017 18:39

FS15, Question 5
There is transitive dependencies: G -> E, I -> H, J so it is not in 3NF
Creating a 3NF:
A -> B, C, D, E, F, I
D -> A
EFI -> G
EI -> H
EI -> J
I -> K
G -> E,I

=>
A -> B, C, D, E, F, G, H, I, J, K
G -> E,I
D -> A

candidate keys: A, D

here, for example, the FD EFI->G does not fulfill any of the requirements for 3NF: it is not trivial, EFI is not a superkey and G is not part of a candidate key.

Synthesis algorithm:

- make schema for every FD: ABCDEFI, DA, EFIG, EIHJ, IK, GEI
- delete the ones included in another: ABCDEFI, EFIG, EIHJ, IK
- first one contains a key, so we have all schemas

https://stackoverflow.com/questions/60174/how-can-i-prevent-sql-injection-in-php SQL injection

Relational Algebra

Freitag, 3. März 2017 08:41

Relational Algebra

- σ Selection
- π Projection
- X Cartesian Product
- ⋈ Join
- ρ Rename
- − Set Minus
- ÷ Relational Division
- ∪ Union
- ∩ Intersection
- ⋉ Semi-Join (left)
- ⋊ Semi-Join (right)
- ⟕ left outer Join
- ⟖ right outer Join

Atoms: basic expressions

a relation in the database
a constant relation

Operators: composite expressions

Operators (composite expressions)

- Selection: $\sigma_p(E_1)$
- Projection: $\Pi_S(E_1)$
- Cartesian Product: $E_1 \times E_2$
- Rename: $\rho_V(E_1)$, $\rho_{A \leftarrow B}(E_1)$
- Union: $E_1 \cup E_2$
- Minus: $E_1 - E_2$

//Recall cartesion Product from earlier in
https://www.systems.ethz.ch/sites/default/files/file/COURSES/2017_SPRING_COURSES/DataModelingAndDatabases/upload/Ch3-RelationalModel.pdf

Selection returns table of relations

Projection

Projection is one of the basic operations of Relational Algebra. It takes a relation and a (possibly empty) list of attributes of that relation as input. It outputs a relation containing only the specified list of attributes *with duplicate tuples removed*. In other words the output must also be a relation.
Aus <http://stackoverflow.com/questions/3461099/what-is-a-projection>

A **projection** is a unary operation written as
where
is a set of attribute names. The result of such projection is defined as the set that is obtained when all tuples in *R* are restricted to the set
Aus <https://en.wikipedia.org/wiki/Relational_algebra#Projection_.28.CF.80.29>

Cartesian Product X

Cartesian Product

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

x

R	
D	E
d_1	e_1
d_2	e_2

=

Result				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_1	b_1	c_1	d_2	e_2
a_2	b_2	c_2	d_1	e_1
a_2	b_2	c_2	d_2	e_2

jedes mit jedem
=> huge result set (n*m)
only useful in combination with a selection -> join

Natural Join ⋈
$R \bowtie S$ combines the the selection of the entries that fulfill R and the entries that fulfill S. Returns a table with all entries that are in at least one of the two relations.

$$R \bowtie S = \Pi_{A1,\dots,Am,\ R.B1,\dots,R.Bk,\ C1,\dots,Cn}(\sigma_{R.B1=S.B1\ \wedge\dots\wedge\ R.Bk=S.Bk}(R \times S))$$

R ⋈ S											
R − S				R ∩ S				S − R			
A_1	A_2	…	A_m	B_1	B_2	…	B_k	C_1	C_2	…	C_n
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

The result of the natural join is the set of all combinations of tuples in *R* and *S* that are equal on their common attribute names
Aus <https://en.wikipedia.org/wiki/Relational_algebra#Projection_.28.CF.80.29>

Threeway natural Join
Same but first join two of the three, and then join the third
$student \bowtie attends \bowtie lecture$

Theta-Join $\bowtie_\Theta$
R and S must have **no** common attributes

Theta-Join

Two Relations:
- R(A1, …, An)
- S(B1, …, Bm)
- A binary operator (<, >, =, ≥, ≤); (if "=", it is an equijoin)
- R and S must have no common attributes

$R \bowtie_\theta S = \sigma_\theta(R \times S)$

$R \bowtie_\theta S$

$R \bowtie_\theta S$							
R				S			
A_1	A_2	…	A_n	B_1	B_2	…	B_m
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

same as natural join but instead of comparing whether tuples are equal to match, you compare it using the binary operator

join variants

left join
keep lines on left side that didn't match on right

natural join (by matching tuples)
combining the lines that fit the same C (end of left line and start of right)

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⋈

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Result				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1

left outer join
natural join AND unmatched tuples from only the left Relation table (these are now in a new line)

- left outer join (natural join + unmatched tuples from L)

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⟕

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Result				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	-	-

right outer join
same but keeping those unmatched from the right side instead of the left in a new line
full outer join
keeping both in new lines if not matched

- (full) outer join

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⟗

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	-	-
-	-	c_3	d_2	e_2

Left/right semi join \ltimes \rtimes
Keep the lines of L/R for which a line in R/L matches

- left semi join (tuples from L matching tuples on R)

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⋉

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat		
A	B	C
a_1	b_1	c_1

Rename operator ρ
renaming of relation names
is needed to process self-joins and recursive relationships
e.g. two-level dependencies ("grandparents")

$$\Pi_{L1.Prerequisite}(\sigma_{L2.\ Follow\text{-}up=5216 \wedge L1.Follow\text{-}up = L2.Prerequisite} (\rho_{L1}(requires) \times \rho_{L2}(requires)))$$

Renaming of attribute names

$$\rho_{Requirement \leftarrow Prerequisite}(requires)$$

A **rename** is a unary operation written as
where the result is identical to *R* except that the *b* attribute in all tuples is renamed to an *a* attribute
Aus <https://en.wikipedia.org/wiki/Relational_algebra#Projection_.28.CF.80.29>

Intersection
only applicable if both relations have the same schema (same attribute names and attribute domains)

$$\Pi_{PersNr}(Lecture) \cap \Pi_{PersNr}(\sigma_{Level=FP}(Professor))$$

28

Relational projections always return distinct tuples so DISTINCT is never needed. Duplicate tuples are not permitted in the RA - that being one major difference between the relational model and the SQL model.

share edit flag

answered Feb 2 '11 at 15:42

sqlvogel
17.3k 1 22 52

Division
$R \div S$ Find lines in R that match with lines in S on their common elements but remove those common elements. (needs to fulfill _all_ in S)

The division is a binary operation that is written as $R \div S$. The result consists of the restrictions of tuples in R to the attribute names unique to R, i.e., in the header of R but not in the header of S, for which it holds that all their combinations with tuples in S are present in R. For an example see the tables *Completed*, *DBProject* and their division:

Completed

Student	Task
Fred	Database1
Fred	Database2
Fred	Compiler1
Eugene	Database1
Eugene	Compiler1
Sarah	Database1
Sarah	Database2

DBProject

Task
Database1
Database2

Completed ***÷*** ***DBProject***

Student
Fred
Sarah

Division can be simulated with other operators:

$$R \div S = \Pi_{(R-S)}(R) - \Pi_{(R-S)}((\Pi_{(R-S)}(R) \times S) - R)$$

Division: Example

$$R \div S = \Pi_{(R-S)}(R) - \Pi_{(R-S)}((\Pi_{(R-S)}(R) \times S) - R)$$

- R = attends; S = Lecture
- Π_{Legi}**(attends)**
  All students (who attend at least one lecture)
- Π_{Legi}**(attends) x Lecture**
  All students attend all lectures
- **(Π_{Legi}(attends) x Lecture) – attends**
  Lectures a student does not attend
- Π_{Legi}**((Π_{Legi}(attends) x Lecture) – attends):**
  Students who miss at least one lecture
- $\Pi_{Legi}(attends) - \Pi_{Legi}((\Pi_{Legi}(attends) \times Lecture) - attends)$
  Students who attend all lectures

SI & S2PL

Samstag, 5. August 2017 18:00

Snapshot Isolation:

For every variable you have written to (it's not about read from variables)
Abort if at commitment time you notice that somebody else already wrote to this variable **since your starting time.** (even if not since your first actual action)

? Wieso abort statt einfach überschreiben? Und wieso nicht auch wenn jemand gelesen hat aborten?

S2PL:

Like 2PL, but only unlock write-locks at commit. Read-locks may be released whenever in the second phase.

Replication

Samstag, 5. August 2017 18:40

Forms of replication

Synchronous

- Advantages:
  - No inconsistencies (identical copies)
  - Reading the local copy yields the most up to date value
  - Changes are atomic
- Disadvantages: A transaction has to update all sites (longer execution time, worse response time)

Update everywhere

- Advantages:
  - Any site can run a transaction
  - Load is evenly distributed
- Disadvantages:
  - Copies need to be synchronized

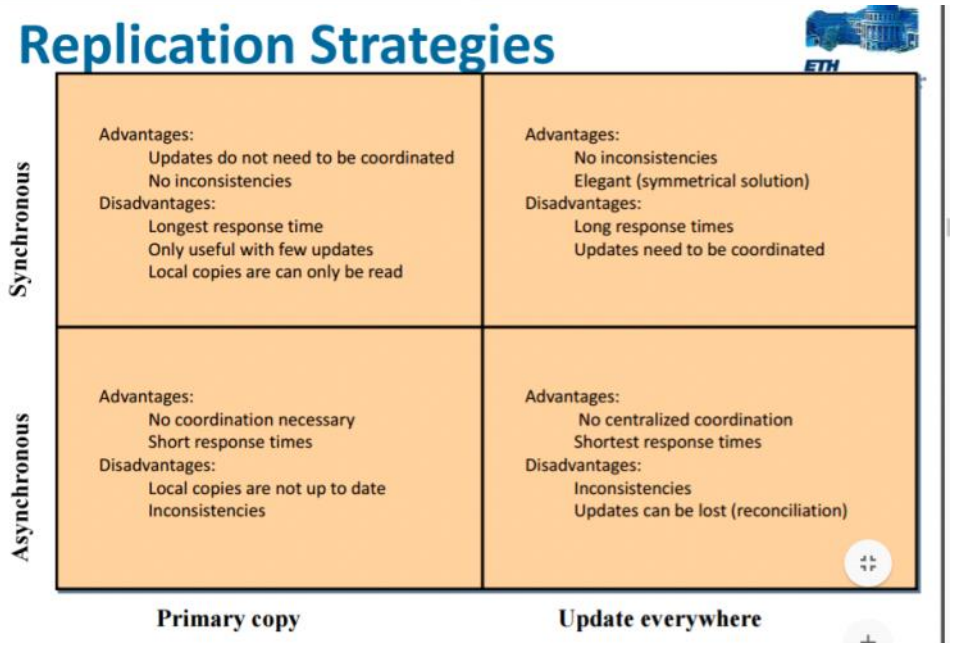
Asynchronous

- Advantages: A transaction is always local (good response time)
- Disadvantages:
  - Data inconsistencies
  - A local read does not always return the most up to date value
  - Changes to all copies are not guaranteed
  - Replication is not transparent

Primary Copy

- Advantages:
  - No inter-site synchronization is necessary (it takes place at the primary copy)
  - There is always one site which has all the updates
- Disadvantages:
  - The load at the primary copy can be quite large
  - Reading the local copy may not yield the most up to date value

Replication Strategies



Exam Prep

Montag, 26. Februar 2018 09:06

Entity Relationship Model
is_a is not complete. A Person does not have to take part in the is_a relationship when the graph says (man, woman) is_a person.
Also, is_a says that 'man' is person. But it does not say that 'man' cannot be 'woman'.

There is no way to say a father cannot be his own son.

If we fix all except one object in a relationship, the relationship is uniquely identified or inexistent. There are no two relationships where A and B are the same but C is not, unless the line to C is annotated 'N' .

Max-Min notation
It is 'how often can the same object participate in the relationship'.
Compared: the usual 1/N notation is 'how many different objects can participate in the relationship'.
min-max syntax: $(0,1)$, $(0,*)$ ...

SQL
JOIN without ON in mySQL generates a cross product. not a natural join. Best to always use ON with postgresql for the DMDB guys.
Similarly, FROM A, B generates a cartesion product.

WHERE … ALL (…)
WHERE … ANY (…)
are a thing

Cannot use an Aggregate in the WHERE clause

Union is distinct

Look at Check, Primary Key, Constraints, ON DELETE CASCATE and so on
Ex7 looks useful for that

Relational Algebra
Left Join ≠ Left outer Join

FDs

Note that the decomposition of R into R1 and R2 is in fact dependency-preserving: (1) is the only FD of R1, (2) and (3) are the FDs of R2, and toghether, they are a minimal basis of the dependencies of R.

Find candidate keys systematically by computing minimal basis

Normal form

- It is always possible to obtain a 3NF design without sacrificing lossless-join or dependency-preservation.

Aus <https://www.cs.sfu.ca/CourseCentral/354/zaiane/material/notes/Chapter7/node12.html>

What is 3NF vs BCNF?

Problems can occur when there is a key consisting of multiple attributes e.g. ABC. If A->B, there is redundancy. If A's are the same for a few tuples, B's are the same, too, so we use a lot of storage, though there is no need for it.

Join Algorithms

Solution

Operator	Minimal buffer size	Maximal buffer size	Replacement policy
Nested-Loop Join	2 pages	all pages of the smaller relation + 1 page of the outer relation	Most-Recently-Used
Grace Hash Join	square root of the number of pages of the smaller relation + 1 page of the outer relation	all pages of the smaller relation + 1 page of the outer relation	-
Sort Merge Join	sorting: 2 pages for the external sort with multiple merge steps or square root of the number of pages of the larger relation if we only perform one merge step in the external sort. merging: 2 pages	all pages of both relations	-
Table Scan	1 page	number of pages that can be read in one I/O request	-
Index Scan	2 pages	entire B-Tree + all pages of the relation	Least-Frequently-Used

For all of these holds that if you want to materialize the output, you will need one additional page to store the output.

Look at **old** exercise 10 solution

What is "Indexed Nested loop Join"?