Database Systems

R4 Cheng

March 6, 2025

Best practice:

1. Always prepare your own key

> Say what you want instead of how to do

Use IN/Not in to check if a value is in a set

Datalog

Each query is a rule.

E.g. For all values of Part, Subpart, and Qty,

if there is a tuple $\langle$Part, Subpart, Qty$\rangle$ in Assembly,

then there must be a tuple $\langle$Part, Subpart$\rangle$ in Components.

```
Components(Part, Subpart) :- Assembly(Part, Subpart, Qty).
```

E.g. For all values of Part, Part2, Subpart, and Qty,

if there is a tuple $\langle$Part, Part2, Qty$\rangle$ in Assembly, **and** a tuple $\langle$Part2, Subpart$\rangle$ in Components,

then there must be a tuple <Part, Subpart>in Components.

```
Components(Part, Subpart) :- Assembly(Part, Part2, Qty),
                             Components(Part2, Subpart).
```

> Each application of a Datalog rule can be understood in terms of relational algebra

Unsafe rules

```
(Unsafe) V(x, y, z) :- Actor(x, y, 1998), z > 200
```

This is unsafe because **z** is not bound to any relation, meaning it can take on infinitely many values.

```
(Unsafe) W(x,y,z) :- Actor(x,y,z), not Plays(t,x)
```

This is unsafe because The variable **t** appears only in the negated literal not Plays(t, x) and does not appear in any positive literal in the body

> Every variable should appear in at least one positive body atom

Relational Algebra

- Defines a set of basic operations on relations
- Each operation returns a relation
- **Result** of an operation can be the input of another operation

Basic operations:

- Selection (σ): Selects a subset of rows from relation.
- Projection (π): Deletes attributes that are not in the projection list and deletes duplicate rows.
- Union ($\cup$): Tuples in relation 1 and in relation 2.
- Set-difference ($-$): Tuples in relation 1 but not in relation 2.
- Cross product ($\times$): Allows us to combine two relations. it returns all possible pairs of tuples from the two relations.
- Rename (ρ): Renames the attributes of a relation. E.g. $\rho_{e1}(Emp)$ renames the relation Emp to e1.

Join is a combination of selection and cross product.

$$R \bowtie S = \sigma_{condition}(R \times S)$$

Relational Calculus

- First-order logic
- Tuple relational calculus (TRC)
- Domain relational calculus (DRC)

Each relational predicate P is:

- Atom (Actor(x, y, z))
- P $\wedge$ P (conjunction)
- P $\vee$ P (disjunction)
- P $\Rightarrow$ P (implication)
- $\neg$ P (negation)

- $\forall$ x P (for all x P holds)
- $\exists$ x P (for an x P holds)

Examples

Exists a schema:

$Movie(\underline{mid}, title, year, total - gross)$
$Actor(\underline{aid}, name, b - year)$
$Plays(\underline{mid}, \underline{aid})$

Q: Actor who played only in movies produced in 1990

$Result(x) = \forall y.Play(y, x) \Rightarrow \exists z \exists t.Movie(y, z, 1990, t)$

Tuple Relational Calculus

Form: $\{T \mid p(T)\}$

The result of this query is the set of all tuples t for which the formula $p(T)$ evaluates to true with $T = t$.

Domain Relational Calculus

Form: $\{< x_1, x_2, \ldots, x_n >\mid p(< x_1, x_2, \ldots, x_n >)\}$, where each x_i is either a domain variable or a constant and $p(< x_1, x_2, \ldots, x_n >)$ denotes a **DRC formula**.

TODO

- fully understand pages, frames, and buffer pool

Storage and Indexing

Heap files

- No order in the file
- new pages inserted at the end of the file

Asymptotic I/O access:

- search: $O(n)$
- insert: $O(1)$ insert at the end
- delete: $O(1)$ after finding the record
- update: $O(1)$ after finding the record

Index

An **index** is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations.

> We use the term **data entry** to refer to the records stored in an index file.

There are three main alternatives for what to store as a data entry in an index:

1. A data entry $k*$ is an actual data record (with searh key value k).
2. A data entry $< k, rid >$ pair, where rid is the record id of a data record with search key value k.
3. A data entry $< k, red - list >$ pair, where $red - list$ is a list of record ids of data records with search key value k.

Alternatives 1 is clustered, while 2 and 3 are can be a clustered index only if the data records are sorted on search key field.

Clustered index: The data records is the same as or close to the ordering of data entries in some index.

Unclustered index: The data records are not ordered according to the index.

- because there is no order in data file, **it must be dense**

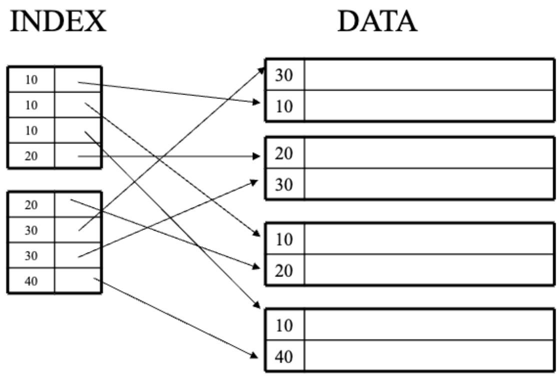


Figure 1: Uncluster Index

1. Hash-based Indexing
2. Tree-based Indexing

Dense vs Sparse Indexes (size vs time trade-off)

- Dense index: each tuple is pointed by an entry in the index.
- Sparse index: each page has an entry in the index.

B+ Tree

- Keeps tree height-balanced
- Search/Update/Insert/Delete $O(\log_f N)$

> f = fanout = the average number of child nodes in an interal node, N = number of leaf pages

0.0.1 Insert

- Pick the proper leaf node and insert the key.
- If the leaf node is full (> 2d keys), split it into two nodes
- Insert the new key in the parent node
- If the parent node is full, split it and so on

TODO: add copy up and push up concept (cowbook p. 350)

What if we need to split the root?

Create a new root with one key and two children (old root)

> This is why root is an exception to the [d, 2d] rule

0.0.2 Delete

Deleted key may remain in the interal node

After deletion, if a node has fewer than d keys, borrow a key from a sibling or merge with a sibling.

If we can borrow a key from a sibling, after borrowing, we need to update the parent node with the new key.

If we cannot borrow a key from a sibling, we need to **merge** the node with a sibling and **remove the dangling key and pointer**.

Hash-based Indexing

0.0.3 Extensible hash index

Allow hash index to grow ⇒ no overflow pages and avoid performance degradation

- Directory: array of pointers to buckets
- Global depth i: number of bits in the directory
- Local depth: number of bits in the hash value

Prons and Cons:

- Now overflow pages ⇒ always one I/O access
- Bucket array may no longer fit in memory

Example: three records whose keys share the first 30 bits. A page split would require setting $i = 30$, i.e., accommodating for 2^30 entries in array! $\Rightarrow$ many useless entries in bucket array!

Linear Hash Index

Attributes:

- Add only one page at a time
- Allow overflow pages
- the number of pages n is no longer a power of 2
- Use the last i bits of the hash value to determine the bucket
- if the last $i \geq n$, change MSB from 1 to 0 (but only in the directory)

Index Selection

Given a query workload and a schema, find the set of indexes that optimize the execution.

The query workload:

- Queries and their frequencies
- Queries are both data retrieval and data manipulation

RDBMS vendors provide wizards:

- AutoAdmin (Microsoft SQL Server)
- SQL Server/ Oracle Index Tuning Wizard
- DB2 Index Advisor

Query Evaluation

Algorithms for selection

Return tuples in Relation R that satisfy predicate $P(A = a, A > a, \dots)$

- Table scan: Read its pages one by one; return tuples that satisfy the predicate.
- Index scan: Use an index on attribute A to find tuples that satisfy the predicate.

Cost of Selection Algorithms

I/O access is the dominant cost

- $B(R)$: number of pages of R
- $|R|$ or $T(R)$: number of tuples in R

Memory requirements: M = #buffers (pages) in the main Memory

Cost of Table-scan

- Cost: $B(R) ==$ Read every page of R once
- Memory requirements: $M > 0$

Cost of Index-scan

- Read only pages of R with tuples that satisfy P
- $S(P)$: fraction of tuples in R that satisfy P
- Cost: $B(R) \times S(P)$ if index is clustered (Tuples with same values of A are in the same page)
- Cost: $T(R) \times S(P)$ if index is unclustered (Tuples with same values of A may be in different pages)
- Memory requirements: $M > 0$

Example:

Consider a relation R with 1000 tuples stored in 100 pages ($B(R) = 100$ and $T(R) = 1000$).

Suppose the predicate P is $Age > 30$ and $S(P) = 0.2$ (20% of tuples satisfy P).

- If the index is clustered:
  - Cost: $B(R) \times S(P) = 100 \times 0.2 = 20$ pages
- If the index is unclustered:
  - Cost: $T(R) \times S(P) = 1000 \times 0.2 = 200$ pages

In this example:

- For a clustered index, only 20 pages need to be read.
- For an unclustered index, 200 pages need to be read due to the scattered nature of the tuples.

Table-scan vs Index-scan

Index-scan is faster than Table-scan if **small fraction** of tuples satisfy the predicate. aka. $S(P) << 1 \Rightarrow$ Large $S(P)\&unclusteredindex$ Index-scan is slower than Table-scan

> Index-scan may read many pages multiple times

TO ASK:

indec scan and selection predicate, about tree of coffee, sell

External Sorting

Two pass, multi-way merge sort

Cost: $2B(R)$ in the first pass $+B(R)$ in the second pass

Memory requirements:

- #pages in each run $\leq$ M (from pass 1)
- #runs $<$ M (from pass 2)
- $B(R) \leq M(M-1)$ or simply $B(R) \leq M^2$

General Multi-Way Merge Sort

> General multi-way merge sort can decrease the use of memory but increase the number of I/Os

Pass 0: Produces $B(R)/M$ level-0 runs

Pass i: Merges $M-1$ runs into one longer run

Number of level-i runs = number of level-(i-1) runs / (M - 1)

For x runs (pass 0), pass 1 = $\frac{x}{M-1}$ runs, pass 2 = $\frac{x}{(M-1)^2}$ runs, ..., pass n = $\frac{x}{(M-1)^n} = 1$ runs

$$n = \log_{M-1}(x)$$

Total number of passes = 1 (pass 0) + $\lceil \log_{M-1}(\frac{B(R)}{M}) \rceil$ (pass-i)

$\Rightarrow$

I/O = number of passes $\times$ $2B(R)$ - 1 (for the last pass) $\Rightarrow O(B(R) \cdot log_M(B(R)))$

Memory requirements: $M > 2$ buffers

> If we have more than 2 buffers, we can decrease the number of I/Os

JOIN Algorithms

In Memory Join

Condition: Both relations fit in main memory

External memory join algorithms

Index Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S where r_i = s_join
        add <r, s> to result
```

Sort-Merge Join

Cost: $5B(R) + 5B(S)$

Memory requirements: $M > 2$

$>$ Exception: If more than M pages of R and S share the same value for join attribute (Cost: $\sim B(R)B(S)$) $\Rightarrow$ use nested loops join. E.g. all students and professors work on one project, we have to join all tuples in these relations.

We can optimize the sort-merge join by performing merge phase of sort and merge phase of join at the same time.

Hash Join

no size restrictions on size of Relation s

disadvantage:

> 3B(R) + 3B(S) possibility =¿ the distribution is not on average =¿ recursive hashing partitioning

Query Optimization

Plan space: the set of all possible execution plans

Reduce the cost of executing a query

- Push selection down (Reduce number of tuples)
- Push projection down (Reduce number of attributes)
- Avoid plans with Cartesian product

$>$ Push projection down is less effective than push selection down

Cost Estimation

Goal of cost estimation of a query plan: Maximize **relative** accuracy

Cost plan = sum of costs of all operators

$$T_{join} + T_{selection} + T_{projection}$$

selectivity factor F: ratio of output size to input size $= \frac{output}{input}$

$>$ The meaning of selectivity factor F is to estimate the filter capacity of a query. if $F \approx 0$, better for the index, $F \approx 1$, use table scan

Selinger Style

$V(R, A)$: Number of distinct values of attribute A in relation R

We assume that attributes and predicates are independent

$$\Rightarrow P(A = 2andB = 1) = P(A = 2)P(B = 1) \neq P(A = 2 \mid B = 1)P(B = 1)(?)$$

For point selection: $S = \sigma_{A=a}(R)$

$T(S)$ ranges from 0 to $T(R) = V(R, A) + 1$
And $F = 1/V(R, A)$

System-R style Plan Search

> a.k.a Selinger style

Dynamic programming button up

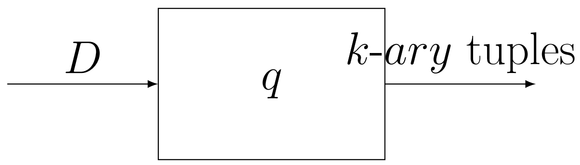
Button up:

- start from the ground relation ($FROM$ syntax)
- build up the plan tree
- count the cost of each subtree

Dynamic programming:

- greedily remove subtrees that costs a lot
- keep only the best plan for each subtree

Query Satisfiability and Equivalence



- D: Database instance
- k-ary tuples: tuples with k attributes
- q: Function on D Over DB S
- $q(D)$ is a k-ary relation on $adom(D)$

> $adom(D)$: the set of all constants appearing in D

Q. Given a graph E (binary relation), is the diameter of E at most 3?

Three Fundamental Query Optimization Problems

1. The Query Satisfiability Problem

Given a query q, is there any database instance D such that $q(D) \neq \emptyset$

```
Select *
from Sells
where price = 10 and price <>10
```

$\Rightarrow \emptyset$

2. The Query Equivalence Problem

Given two queries q and q', of the same arity, is it the case that $q \equiv q'$? i.e. for every database instance D, we have that $q(D) = q'(D)$

```
Sells(sname, cname, price)
    select sname, cname
    from Sells, Sells
    where Sell.sname = Sells.sname and
        Sells.cname = Sells.cname
=
    Select sname, cname
    From Sells
```

3. The Query Containment Problem

Given two queries q and q', of the same arity, is it the case that $q \subseteq q'$? i.e. for every database instance D, we have that $q(D) \subseteq q'(D)$

solve 2 can solve 3 or solve 3 can solve 2? (to check)

The question we want to address are:

1. How can we measure the precise difficulty of these problem?
2. Are there good algorithms for solving these problems?
3. If not, any special cases of these problems for which good algorithm exist?

Turing

Ruduction Method is a common way to prove a problem is not recursive aka. undecidable.

The Reduction Method

$L \leq L^*$ means

- It exists a reduction of L to L^*
- Aka. L^* is at least as hard as L
- If L is undecidable then L^* is undecidable

- This relationship is transitive

Conjunctive Query

Def. $\Pi_x(\sigma_\theta(R_1 \times ..., \times R_n))$, where θ is a conjunction of equlity atomic formulas (equijoin)

```
Q(X):-
```

Homomorphism

Compress the query to a smaller query

Homomorphism between queries

Proof of Homomorphism

what c stand for of $c(x_1)$? c is mapping

understaind connel database!!

$$Q_1 \in Q_2 \Rightarrow h : Q_2 \rightarrow Q_1$$

```
Q_1(x_1, x_2) :- R(x_1, x_3, x_2), R(x_1, x_3, x_2)
Q_2(y_1, y_2): R(y_1, y_4, y_2)
```

Concurrency

Transection: A 'program' of database operations

> I/O activity can be done in parallel with CPU activity in a computer (cowbook 16.3.1)

- Atomicity: All actions in the transaction happen or none happen.
- Consistency: If each transaction is consistent and the database starts in a consistent state before the transaction begins, then the database will be consistent when the transaction ends.
- Isolation: Execution of a transaction is isolated from other transactions.
- Durability: Once a transaction is committed, its effects persist.

Serializability

Conflict Equivalence of Two Schedules:

1. They both involve the same set of actions of the same transactions respectively.
2. The order of every pair of conflicting actions of two transactions is the same in both schedules.

Locking

Goal of it: 1. Guarantee serializability, 2. preserve high concurrency

Questions to ask:

- What modes of locks to provide?
- How to **get** and **release** locks? $\Rightarrow$ what sequence, how long?
- What units to lock? *Database* $\Rightarrow$ *Relation* $\Rightarrow$ *pages* $\Rightarrow$ *tuples* $\Rightarrow$ *attributes*

Unlocking: Release all relevant locks at once or leaf ot root

Two-Phase Locking

- Growing Phase: A transaction may obtain locks but may not release any lock.
- Shrinking Phase: A transaction may release locks but may not obtain any new lock.

2PL does not allow the swap of conflicting operations ($\Rightarrow$ serial order); and it is possible to swap non-conflicting operations ($\Rightarrow$ high concurrency)

Problem: it might cause cascading rollback

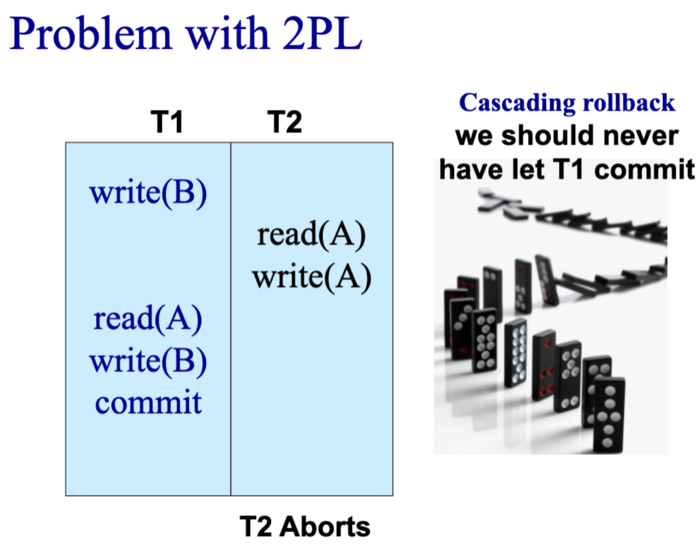


Figure 2: LSNs

$T1$ reads the udpated data A of $T2$ and is committed, but $T2$ is aborted afterwards. $\Rightarrow$ $T1$ must be rolled back, aka. cascading rollback

Solution $\Rightarrow$ strict 2PL (but reduce concurrency)

Strict Two-Phase Locking

1. If a transection T wants to read / modify an object, it first requests a shared / exclusive lock on the object.
2. All locks held by a transaction are released when the transaction is completed.

> A transaction that has an exclusive lock can also read the object.

state that cannot result from any serial execution of the three transactions

Intention Locks

- Use the put on the parent level to indicate that the child level will be locked
- IS, IX, SIX

> SIX allows the read at current level and says that the transaction will write at a lower level.

Compatibility rule: Reject intention lock if it allows incompatible licks on data items

Degrees of Consistency

- Degree 0: T does not overwite the dirty data of other transactions (short X lock)
- Degree 1: T does not commit any writes until the end of the transaction (long X lock)
- Degree 2: T does not read dirty data from other transactions (long X lock and short S lock)
- Degree 3: Other T_x do not diry any data read by T before T commits (long X lock and long S lock)

Crash Recovery

Stealing Frames and Forcing Pages

A page might be written to disk before the transaction T_1 is committed. E.g. when the buffer pool is full and an another transaction T_2 needs to **bring** in a page, the buffer mangeer might choose to replace the frame. (T_2 steals a frme from T_1)

⇒ need to **undo** the changes

> Of course, that frame must be unpined by T_1, i.e. T_1 temporarily does not need the frame.

After a transaction is committed, all changes of an object has immediately been written to disk. This is force approach

⇒ need to **redo** the changes

ARIES

ARIES is a **recovery algorithm** designed to work with a steal, no-force approach.

The restart process:

- Analysis
- Redo
- Undo

The Log

Other Log-Related Structure

1. Transaction Table: It contains one entry for each active transaction. It contains xid, transaction state, **lastLSN**, and others

> entry: a record for a active transaction.

> lastLSN: the LSN of the most recent log record for the transaction.

2. Dirty page Table: It contains one entry for each dirty page in the buffer pool.

The entry contains a field **rescLSN**, which is the LSN of the first log record that caused the apge to become dirty.

The Write-Ahead Log Protocol

Two basic rules:

1. Before any changes are written to disk, all log records describing these changes MUST first be written to stable storage. (Write)
2. A transaction cannot be committed until all log records have been written to stable storage. (Ahead)

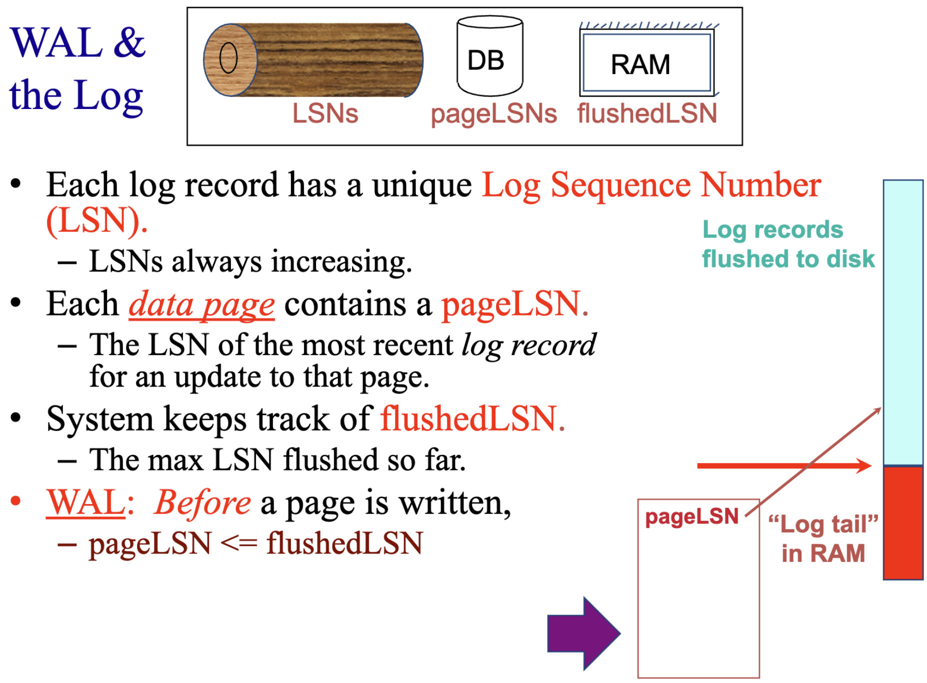


Figure 3: LSNs

- All log records have a unique log sequence number (LSN) which is monotonically increasing.
- (In Database) All data page include a page LSN which is the LSN of the last log record that modified the page.
- (In RAM) The system keeps tracking of flushedLSN which is the largest LSN of all log records that have been written to disk.

The $pageLSN \leq flushedLSN$ is the core rule of WAL, ensuring that all log records are written to disk before the corresponding data pages. $\Rightarrow$ keep data consistent and recoverable.

1

TO TA/Professor

- why in 2 pass merge sort the number of M unit increases?
- book 9.7 why to point free space there?
- why not small relation in memory and scan the large relation and join $\Rightarrow O(T(S)B(R))$
- for clustered relation, Index nested loops is better than Optimized Two-pass multi-way merge sort?