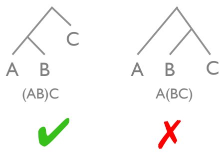
Selinger Optimizer

parens(N) = 1
Only left-deep plans
ensures pipelining



Dynamic Programming
Idea: If considering ((ABC)DE)
compute best (ABC), cache, and reuse
figure out best way to combine with (DE)

```
Dynamic Programming Algorithm
   compute best join size 1, then size 2, …
```
~$O(N*2^N)$

Summary

Single operator optimizations
- Access paths
- Primary vs secondary index costs
- Projection/distinct
- Predicate/project push downs

2 operators aka Joins
- Nested loops, index nested loops, sort merge

Full plan optimizations
- Naïve vs Selinger join ordering

Selectivity estimation
- Statistics and simple models

Summary

Query optimization is a deep, complex topic
Pipelined plan execution
Different types of joins
Cost estimation of single and multiple operators
Join ordering is hard!

You should understand

Estimate query cardinality, selectivity
Apply predicate push down
Given primary/secondary indexes and statistics,
- pick best index for access method + est cost
- pick best index for join + est cost
- pick cheaper of two execution plans

Given 3 or 4-way join
- estimate join selectivity
- pick the best join order

Transactions, Concurrency, Recovery

Transfer $1000 from Evan to Neha

Check if Evan has $1000
Evan's Account -= $1000
Neha's Account += $1000

Transfer $1000 from Evan to Neha

Check if Evan has $1000

Evan's Account -= $1000

~~Neha's Account += $1000~~

Program crash or user presses cancel:

Money disappeared

Transfer $1000 from Evan to Neha

Check if Evan has $1000

~~Evan's Account -= $1000~~

~~Neha's Account += $1000~~

OOPS! Not enough money

Two transfers: Starting with $1500

Check if Evan has $1000

    Check if Evan has $1000

Evan's Account -= $1000

    Evan's Account -= $1000

Negative balance!

Neha's Account += $1000

    Eugene's Account += $1000

Transactions

Sequence of actions treated as a single unit

Atomicity: Apply all changes or none

("atomic" because it is indivisible)

Solves the crash problem

Isolation: Illusion that each transaction executes sequentially, without concurrency

Transaction Guarantees

Atomicity

- "all or nothing": All changes applied, or none are
- users never see in-between transaction state

Consistency

- database always satisfies Integrity Constraints
- Transactions move from valid database to valid database

Isolation:

- from transaction's point of view, it's the only one running

Durability:

- if transaction commits, its effects *must persist*

Transactions

Transaction: a sequence of actions

- action = read object, write object, commit, abort
- API between app semantics and DBMS's view

User's view

T1: begin	A=A+100	B=B-100	END
T2: begin	A=A-50	B=B+50	END

DBMS's logical view

T1: begin	r(A) w(A)	r(B) w(B)	END
T2: begin	r(A) w(A)	r(B) w(B)	END

Concepts

Concurrency Control

techniques to ensure correct results when running transactions concurrently

what does this mean?

Recovery

On crash or abort, how to get back to a consistent (correct) state?

The two are intertwined!

What is Correct?

Serializability

Regardless of the interleaving of operations, result same as a serial ordering

Schedule

One specific interleaving of the operations

Serial Schedules

Logical xacts

T1: r(A) w(A) r(B) w(B) (e.g. A=A+100;B=B-100)
T2: r(A) w(A) r(B) w(B) (e.g. A=A*1.5; B=B*1.5)

No concurrency (serial 1)

T1: r(A) w(A) r(B) w(B)
T2: r(A) w(A) r(B) w(B)

No concurrency (serial 2)

T1: r(A) w(A) r(B) w(B)
T2: r(A) w(A) r(B) w(B)

Are serial 1 and serial 2 equivalent?

More Example Schedules

Logical xacts e.g. A=0 B=0

T1: r(A) w(A) r(A) w(B) e.g. A=A+1; B=A+1
T2: r(A) w(A) r(B) w(B) e.g. A=A+10; B=B+1

Concurrency (bad)

T1: r(A) w(A) r(A) w(B)
T2: r(A) w(A) r(B) w(B)

Concurrency (same as serial T1,T2!)

T1: r(A) w(A) r(A) w(B)
T2: r(A) w(A) r(B) w(B)

Concepts

Serial schedule

One transaction at a time. no concurrency.

Equivalent schedule

the database state is the same at end of both schedules

Serializable schedule (gold standard)

equivalent to a serial schedule

SQL → R/W Operations

```
UPDATE   accounts
SET      bal = bal + 1000
WHERE    bal > 1M
```

Read all balances for every tuple

Update those with balances > 1M

Does the access method mater?

Why Serializable Schedule? Anomalies

Reading in-between (uncommitted) data

T1: R(A) W(A) R(B) W(B) abort
T2: R(A) W(A) commit

WR conflict or dirty reads

Reading same data gets different values

T1: R(A) R(A) W(A) commit
T2: R(A) W(A) commit

RW conflict or unrepeatable reads

Why Serializable Schedule? Anomalies

Stepping on someone else's writes

T1: W(A) W(B) commit
T2: W(A) W(B) commit

WW conflict or lost writes

Notice: all anomalies involve writing to data that is read/written to.

If we track our writes, maybe can prevent anomalies

Conflict Serializability

What is a conflict?

For 2 operations, if run in different order, get different results

Conflict?	R	W
R	**NO**	YES
W	YES	YES

Conflict Serializability

def: a schedule that is conflict equivalent to a serial schedule

Meaning: you can swap non-conflicting operations to derive a serial schedule.

$\forall$ conflicting operations O1 of T1, O2 of T2

O1 always before O2 in the schedule or

O2 always before O1 in the schedule