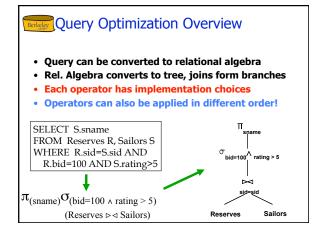


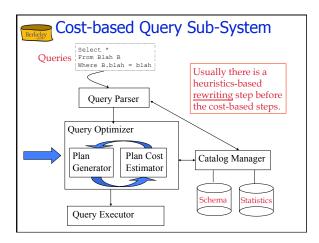


- · Choice of single-table operations
  - Depends on indexes, memory, stats,...
- Joins
  - Blocked nested loops:
    - simple, exploits extra memory
  - Indexed nested loops:
    - · best if 1 rel small and one indexed
  - Sort/Merge Join
    - good with small amount of memory, bad with duplicates
  - Hash Join
    - fast (enough memory), bad with skewed data
- These are "rules of thumb"
  - On their way to a more principled approach...





- <u>Plan:</u> Tree of R.A. ops (and some others) with choice of algorithm for each op.
  - Recall: Iterator interface (next()!)
- · Three main issues:
  - For a given query, what plans are considered?
  - How is the cost of a plan estimated?
  - How do we "search" in the "plan space"?
- Ideally: Want to find best plan.
- Reality: Avoid worst plans!



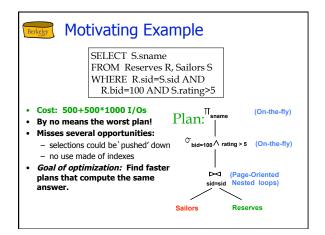
Let's go through some examples

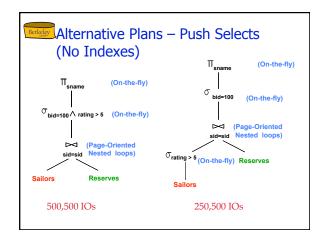
• Just to get a flavor...

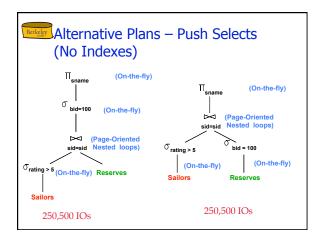


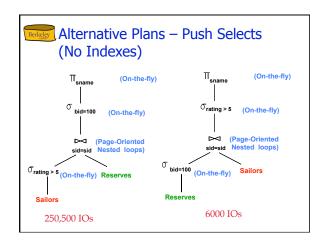
Sailors (<u>sid: integer</u>, sname: string, rating: integer, age: real) Reserves (<u>sid: integer</u>, bid: integer, day: dates, rname: string)

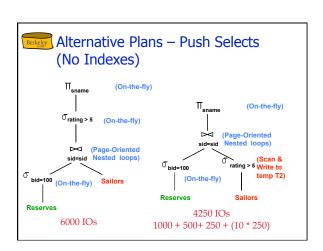
- · As seen in previous lectures...
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
  - Assume there are 100 boats
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.Assume there are 10 different ratings
- Assume we have 5 pages in our buffer pool!

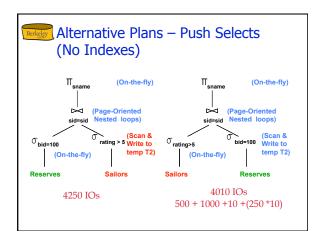


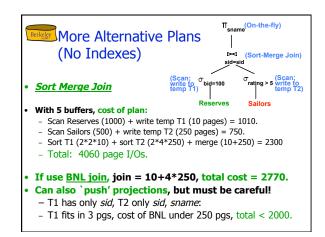


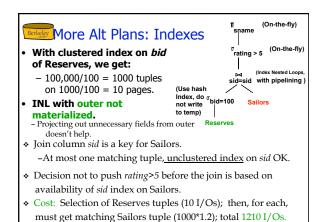


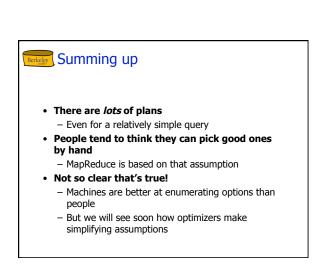




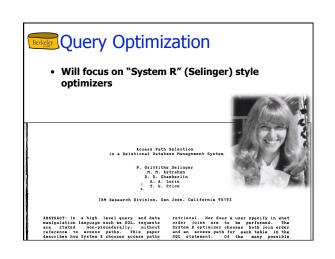














# Highlights of System R Optimizer

#### • Impact:

- Most widely used currently; works well for 10-15 joins.

#### · Cost estimation:

- Very inexact, but works ok in practice.
- Statistics in system catalogs used to estimate cost of operations and result sizes.
- Considers combination of CPU and I/O costs.
- System R's scheme has been improved since that time.

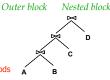
#### · Plan Space: Too large, must be pruned.

- Many plans share common, "overpriced" subtrees • ignore them all!
- In some implementations, only the space of left-deep plans is considered.
- Cartesian products avoided in some implementations.

## Query Blocks: Units of Optimization

- Break query into query blocks
- Optimized one block at a time
- Uncorrelated nested blocks computed once
- Correlated nested blocks like function calls
  - But sometimes can be "decorrelated"
- Beyond the scope of CS186!
- . For each block, the plans considered are:
  - All available access methods, for each relation in FROM clause.
  - All left-deep join trees
    - right branch always a base table
    - consider all join orders and join methods

SELECT S.sname FROM Sailors S WHERE S.age IN (SELECT MAX (S2.age FROM Sailors S2 GROUP BY S2.rating)





# Schema for Examples

Sailors (sid: integer, sname: string, rating: integer, age: real) Reserves (sid: integer, bid: integer, day: dates, rname: string)

#### Reserves:

- Each tuple is 40 bytes long, 100 tuples per page, 1000 pages. 100 distinct bids.

#### Sailors:

Each tuple is 50 bytes long, 80 tuples per page, 500 pages. 10 ratings, 40,000 sids.



### Translating SQL to Relational Algebra

SELECT S.sid, MIN (R.day) FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = "red" GROUP BY S.sid

HAVING COUNT (\*) >= 2

For each sailor with at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.



#### Translating SQL to Relational Algebra

SELECT S.sid, MIN (R.day) FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = "red" GROUP BY S.sid HAVING COUNT (\*) >= 2

 $\pi_{\text{S.sid, MIN(R.day)}}$ (HAVING <sub>COUNT(\*)>2</sub> ( GROUP BY S.Sid (  $\sigma_{\text{B.color} = \text{``red''}}$  ( Sailors ⋈ Reserves ⋈ Boats))))



# Relational Algebra Equivalences

- Allow us to choose different join orders and to `push' selections and projections ahead of joins.
- - $\sigma_{c1 \land ... \land cn}(R) \equiv \sigma_{c1}(...(\sigma_{cn}(R))...)$  (cascade)  $\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$ (commute)
- Projections:
- $\pi_{a1}(R) = \pi_{a1}(...(\pi_{a1,...,an}(R))...)$  (cascade)
- **Cartesian Product** 
  - $-R \times (S \times T) \equiv (R \times S) \times T$  (associative)
  - $-R \times S \equiv S \times R$ (commutative)
  - This means we can do joins in any order.
    - · But...beware of cartesian product!



# More Equivalences

- · Eager projection
  - Can cascade and "push" some projections thru selection
  - Can cascade and "push" some projections below one side of a join
  - Rule of thumb: can project anything not needed "downstream"
- · Selection on a cross-product is equivalent to a join.
  - If selection is comparing attributes from each side
- A selection on attributes of R commutes with R S.
  - i.e.,  $\sigma(R \bowtie S)$  ≡  $\sigma(R) \bowtie S$
  - but only if the selection doesn't refer to S!



### **Cost Estimation**

- For each plan considered, must estimate total cost:
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
    - We've already discussed this for various operators - sequential scan, index scan, joins, etc.
  - Must estimate size of result for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.
  - In System R, cost is boiled down to a single number consisting of #I/O + CPU-factor \* #tuples
- Q: Is "cost" the same as estimated "run time"?



#### Statistics and Catalogs

- · Need info on relations and indexes involved.
- Catalogs typically contain at least:

# of tuples in a table (cardinality) NTuples NPages # of disk pages in a table Low/High min/max value in a column Nkeys # of distinct values in a column IHeight the height of an index INPages # of disk pages in an index

- · Catalogs updated periodically.
  - Too expensive to do continuously
  - Lots of approximation anyway, so a little slop here is ok.
- · Modern systems do more
  - Esp. keep more detailed information on data values
    - e.g., histograms



#### Size Estimation and Selectivity

SELECT attribute list FROM relation list WHERE term1 AND ... AND termk

- Max output cardinality = product of input cardinalities
- Selectivity (sel) associated with each term
  - reflects the impact of the *term* in reducing result size.
  - |output| / |input|

Result cardinality = Max # tuples \*  $\prod$  sel,

- Book calls selectivity "Reduction Factor" (RF)
- **Avoid confusion:** 
  - "highly selective" in common English is opposite of a high selectivity value (|output|/|input| high!)



#### Result Size Estimation

- Result cardinality = Max # tuples \* product of all RF's.
- Term col=value (given Nkeys(I) on col) RF = 1/NKeys(I)
- Term col1=col2 (handy for joins too...) RF = 1/MAX(NKeys(I1), NKeys(I2))
- Term col>value

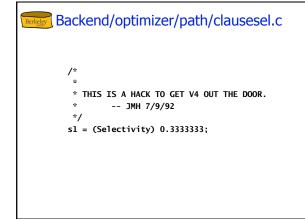
RF = (High(I)-value)/(High(I)-Low(I))

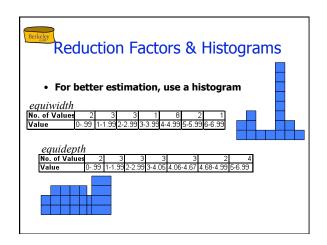
Implicit assumptions: values are uniformly distributed and terms are independent!

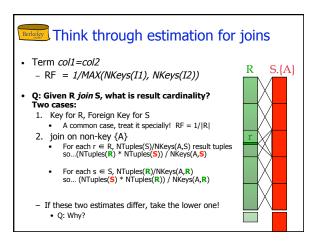
Note, if missing the needed stats, assume 1/10!!!

### Postgres 8: include/utils/selfuncs.h

- /\* default selectivity estimate
   for equalities such as "A = b"
   \*/
- #define DEFAULT EQ SEL 0.005
- default selectivity estimate for inequalities such as "A < b" \*/
- /\* default selectivity estimate for range inequalities "A > b AND A < c" \*/ #define DEFAULT\_RANGE\_INEQ\_SEL 0.005
- /\* default selectivity estimate
   for pattern-match operators
   such as LIKE \*/
  #define DEFAULT\_MATCH\_SEL 0.005
- /\* default number of distinct values in a table \*/ #define DEFAULT\_NUM\_DISTINCT 200
- /\* default selectivity estimate for boolean and null test nodes \*/ #define DEFAULT\_UNK\_SEL 0.005
- #define DEFAULT\_NOT\_UNK\_SEL (1.0 DEFAULT\_UNK\_SEL)









- There are two main cases:
  - Single-relation plans (base case)
  - Multiple-relation plans (induction)
- Single-table queries include selects, projects, and grouping/aggregate ops:
  - Consider each available access path (file scan / index)
    - Choose the one with the least estimated cost
  - Selection/Projection done on the fly
  - Result pipelined into grouping/aggregation



- Index I on primary key matches selection:
  - Cost is Height(I)+1 for a B+ tree.
- Clustered index I matching one or more selects:
  - (NPages(I)+NPages(R)) \* product of RF's of matching selects.
- Non-clustered index I matching one or more selects:

   (NPages(I)+NTuples(R)) \* product of RF's of matching selects.
- Sequential scan of file:
  - NPages(R).
- Recall: Must also charge for duplicate elimination if required



SELECT S.sid FROM Sailors S WHERE S.rating=8

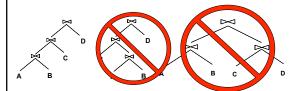
- If we have an index on rating:
  - Cardinality = (1/NKeys(I)) \* NTuples(R) = (1/10) \* 40000 tuples
  - Clustered index: (1/Nkeys(I)) \* (NPages(I) + NPages(R))
    = (1/10) \* (50+500) = 55 pages are retrieved. (This is the *cost*.)
  - Unclustered index: (1/NKeys(I)) \* (NPages(I)+NTuples(R))
  - = (1/10) \* (50+40000) = 4005 pages are retrieved.

#### If we have an index on sid:

- Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500, with unclustered index, 50+40000.
- Doing a file scan:
  - We retrieve all file pages (500).

# Queries Over Multiple Relations

- A System R heuristic: only left-deep join trees considered.
  - Restricts the search space
  - Left-deep trees allow us to generate all fully pipelined plans.
    - · Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., SM join).





# Enumeration of Left-Deep Plans

- · Left-deep plans differ in
  - the order of relations
  - the access method for each relation
  - the join method for each join.
- Enumerated using N passes (if N relations joined):
  - Pass 1: Find best 1-relation plan for each relation.
  - Pass i: Find best way to join result of an (i -1)-relation plan (as outer) to the i'th relation. (i between 2 and N.)
- For each subset of relations, retain only:
  - Cheapest plan overall, plus
  - Cheapest plan for each interesting order of the tuples.



### The Dynamic Programming Table

Subset of tables in FROM clause	Interesting- order columns	Best plan	Cost
{R, S}	<none></none>	hashjoin(R,S)	1000
{R, S}	<r.a, s.b=""></r.a,>	sortmerge(R,S)	1500



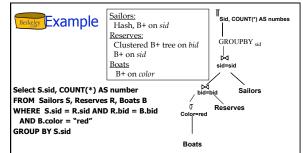
# A Note on "Interesting Orders"

- · An intermediate result has an "interesting order" if it is sorted by any of:
  - ORDER BY attributes
  - GROUP BY attributes
  - Join attributes of *yet-to-be-added* (downstream)



# Enumeration of Plans (Contd.)

- Match an i-1 way plan with another table only if
  - a) there is a join condition between them, or
  - b) all predicates in WHERE have been used up.
  - i.e., avoid Cartesian products if possible.
- · ORDER BY, GROUP BY, aggregates etc. handled as a final step
  - via `interestingly ordered' plan if chosen (free!)
  - or via an additional sort/hash operator
- . Despite pruning, this is exponential in #tables.
- · Recall: in practice, COST considered is #IOs + factor \* CPU Inst



- Pass1: Best plan(s) for accessing each relation
  - Reserves, Sailors: File Scan
  - Q: What about Clustered B+ on Reserves.bid???
  - Boats: B+ tree on color



#### Pass 1

- · Find best plan for each relation in isolation:
  - Reserves, Sailors: File Scan - Boats: B+ tree on color



#### Pass 2

- For each plan in pass 1, generate plans joining another relation as the inner, using all join methods (and matching inner access methods)
  - File Scan Reserves (outer) with Boats (inner)
  - File Scan Reserves (outer) with Sailors (inner)
  - File Scan Sailors (outer) with Boats (inner)
  - File Scan Sailors (outer) with Reserves (inner) - Boats Btree on color with Sailors (inner)
  - Boats Btree on color with Reserves (inner)
- Retain cheapest plan for each (pair of relations, order)



### Pass 3 and beyond

- Using Pass 2 plans as outer relations, generate plans for the next join
  - E.g. Boats B+-tree on color with Reserves (bid) (sortmerge) inner Sailors (B-tree sid) sort-merge
- Then, add cost for groupby/aggregate:
  - This is the cost to sort the result by sid, unless it has already been sorted by a previous operator.
- · Then, choose the cheapest plan



### Physical DB Design

- Query optimizer does what it can to use indices, clustering etc.
- · DataBase Administrator (DBA) is expected to set up physical design well
- Good DBAs understand query optimizers very well



### One Key Decision: Indexes

- · Which tables
- Which field(s) should be the search key?
- Multiple indexes?
- Clustering?



#### Index Selection

- One approach:
  - Consider most important queries in turn.
  - Consider best plan using the current indexes
  - See if better plan is possible with an additional index.
  - If so, create it.
- But consider impact on updates!
  - Indexes can make queries go faster, updates slower.
  - Require disk space, too.



#### Issues to Consider in Index Selection

- Attributes mentioned in a WHERE clause are candidates for index search keys.
  - Range conditions are sensitive to clustering
  - Exact match conditions don't require clustering
    - Or do they???? :-)
- · Choose indexes that benefit many queries
- NOTE: only one index can be clustered per relation!
  - So choose it wisely!



SELECT E.ename, D.mgr FROM Emp E, Dept D WHERE E.dno=D.dno AND D.dname='Toy'

- B+ tree index on *D.dname* supports 'Toy' selection.
  - Given this, index on D.dno is not needed.
- B+ tree index on E.dno allows us to get matching (inner) Emp tuples for each selected (outer) Dept tuple.
- What if WHERE included: `` ... AND E.age=25"?
  - Could retrieve Emp tuples using index on *E.age*, then join with Dept tuples satisfying dname selection.
    - Comparable to strategy that used E.dno index.
  - So, if *E.age* index is already created, this query provides much less motivation for adding an *E.dno* index.



SELECT E.ename, D.mgr FROM Emp E, Dept D WHERE E.sal BETWEEN 10000 AND 20000 AND E.hobby='Stamps' AND E.dno=D.dno

- All selections are on Emp so it should be the outer relation in any Index NL join.
  - Suggests that we build a B+ tree index on D.dno.
- What index should we build on Emp?
  - B+ tree on E.sal could be used, OR an index on E.hobby could be used.
  - Only one of these is needed, and which is better depends upon the selectivity of the conditions.
    - As a rule of thumb, equality selections more selective than range
- Have to understand optimizers to get this right!



## Examples of Clustering

B+ tree index on E.age can be used

- to get qualifying tuples. - How selective is the condition?
- Is the index clustered?
- Consider the GROUP BY query.
  - If many tuples have *E.age* > 10, using E.age index and sorting the retrieved tuples may be costly.
  - Clustered E.dno index may be better!
- **Equality queries and duplicates:** 
  - Clustering on E.hobby helps!

SELECT E.dno, COUNT (\*) FROM Emp E WHERE E.age>10 GROUP BY E.dno

SELECT E.dno

FROM Emp E WHERE E.age>40

SELECT E.dno FROM Emp E WHERE E.hobby=Stamps

# Index-Only Plans

 Answer query without going to heap file!

<E.dno> <E.dno,E.eid>

SELECT D.mgr FROM Dept D, Emp E WHERE D.dno=E.dno

SELECT D.mgr, E.eid FROM Dept D, Emp E WHERE D.dno=E.dno

<E.dno>

SELECT E.dno, COUNT(\*) FROM Emp E GROUP BY E.dno

<E.dno, E.sal> B-tree trick!

SELECT E.dno, MIN(E.sal) FROM Emp E GROUP BY E.dno

<E.sal, E.age>

<E. age, E.sal> | SELECT AVG(E.sal) FROM Emp E WHERE E.age=25 AND E.sal BETWEEN 3000 AND 5000



#### Horizontal Decompositions

- Typical decomposition: Relation is replaced by collection of relations that are projections. Most important case.
  - We will talk about this at length as part of Conceptual DB Design
- Sometimes, might want to replace relation by a collection of relations that are selections.
  - Each new relation has same schema as original, but subset
  - Collectively, new relations contain all rows of the original.
  - Typically, the new relations are disjoint.



#### Horizontal Decompositions (Contd.)

- Contracts (<u>Cid</u>, Sid, Jid, Did, Pid, Qty, Val)
- Suppose that contracts with value > 10000 are subject to different rules.
  - So queries on Contracts will often say WHERE val>10000.
- One approach: clustered B+ tree index on the val field.
- Second approach: replace contracts by two new relations, LargeContracts and SmallContracts, with the same attributes (CSJDPQV).
  - Performs like index on such queries, but no index overhead.
  - Can build clustered indexes on other attributes, in addition!

#### Masking Conceptual Schema Changes

CREATE VIEW Contracts(cid, sid, jid, did, pid, qty, val) AS SELECT

FROM LargeContracts UNION

SELECT \*

FROM SmallContracts

- **Horizonal Decomposition from above**
- Masked by a view.
  - NOTE: gueries with condition val>10000 must be asked wrt LargeContracts for efficiency: so some users may have to be aware of change.
    - I.e. the users who were having performance problems
    - Arguably that's OK -- they wanted a solution!



### Index Tuning "Wizards"

- Both IBM's DB2 and MS SQL Server have automated index advisors
  - Some info in Section 20.6 of the book
- Basic idea:
  - They take a workload of queries
    - · Possibly based on logging what's been going on
  - They use the optimizer cost metrics to estimate the cost of the workload over different choices of sets of indexes
  - Enormous # of different choices of sets of indexes:
    - · Heuristics to help this go faster



#### Tuning Queries and Views

- If a query runs slower than expected, check if an index needs to be re-clustered, or if statistics are too old.
- Sometimes, the DBMS may not be executing the plan you had in mind. Common areas of weakness:
  - Selections involving null values (bad selectivity estimates)
  - Selections involving arithmetic or string expressions (ditto)
  - Selections involving OR conditions (ditto)
  - Complex subqueries (more on this later)
  - Lack of evaluation features like index-only strategies or certain join methods or poor size estimation.
- Check the plan that is being used! Then adjust the choice of indexes or rewrite the query/view.

  – E.g. check via POSTGRES "Explain" command

  - Some systems rewrite for you under the covers (e.g. DB2)
    - · Can be confusing and/or helpful!



#### More Guidelines for Query Tuning

- Minimize the use of DISTINCT: don't need it if duplicates are acceptable, or if answer contains a key.
- Minimize the use of GROUP BY and HAVING:

SELECT MIN (E.age) FROM Employee E GROUP BY E.dno HAVING E.dno=102

SELECT MIN (E.age) FROM Employee E WHERE E.dno=102

\* Consider DBMS use of index when writing arithmetic expressions: E.age=2\*D.age will benefit from index on *E.age*, but might not benefit from index on *D.age*!



#### Guidelines for Query Tuning (Contd.)

Avoid using intermediate relations:

FROM Emp E, Dept D WHERE E.dno=D.dno AND D.mgrname='Joe'

SELECT \* INTO Temp

SELECT E.dno, AVG(E.sal) FROM Emp E, Dept D WHERE E.dno=D.dno AND D.mgrname='Joe' GROUP BY E.dno

and SELECT T.dno, AVG(T.sal) FROM Temp T GROUP BY T.dno

- Does not materialize the intermediate reln Temp.
- ❖ If there is a dense B+ tree index on <dno, sal>, an index-only plan can be used to avoid retrieving Emp tuples in the left query!



- Want to understand DB design (tables, indexes)?
  - Must understand query optimization
- Two parts to optimizing a query:
  - Consider a set of alternative plans, pruning search
    - E.g., left-deep plans only
    - avoid Cartesian products.
    - Prune plans with *interesting orders* separate from unordered plans
  - Must estimate cost of each plan that is considered.
    - Output cardinality and cost for each plan node.
    - Key issues: Statistics, indexes, operator implementations.



#### • Single-relation queries:

- All access paths considered, cheapest is chosen.
- - Selections that *match* index
  - whether index key has all needed fields
  - whether index provides tuples in an interesting order.



### More Points to Remember

#### • Multiple-relation queries:

- All single-relation plans are first enumerated.
  - Selections/projections considered as early as possible.
- Use best 1-way plans to form 2-way plans. Prune losers.
- Use best (i-1)-way plans and best 1-way plans to form iway plans
- At each level, for each subset of relations, retain:
  - best plan for each interesting order (including no order)



- · Optimization is the reason for the lasting power of the relational system
- But it is primitive in some ways
- New areas: many!
  - Smarter summary statistics (fancy histograms and "sketches")
  - Auto-tuning statistics,
  - Adaptive runtime re-optimization (e.g. eddies),
  - Multi-query optimization,
  - And parallel scheduling issues, etc.