Database Management Systems

Fall 2001

CMPUT 391: Query Processing & Optimization

Dr. Osmar R. Zaïane

University of Alberta

Chapters 12, 13 &16 of Textbook

Course Content

- Introduction
- Database Design Theory
- **Query Processing and Optimisation**
- Concurrency Control
- Data Base Recovery and Security
- Object-Oriented Databases
- Inverted Index for IR
- XML
- Data Warehousing
- Data Mining
- Parallel and Distributed Databases
- Other Advanced Database Topics

Objectives of Lecture 3

Query Processing and Optimization

- Get a glimpse on query processing and evaluation.
- Introduce the issue of query planning and plan selection.
- Understand the importance of good database design for good performance.

Query Processing and Optimization

- Query Processing and Planning
- System Catalog
- Evaluation of Relational Operations
- Cost Estimation and Plan Selection
- Physical Database Design Issues
- Database Tuning

Overview of Query Processing

- The aim is to transform a query in a high-level declarative language (SQL) into a correct and efficient execution strategy
- Query Decomposition
  - Analysis
  - Conjunctive and disjunctive normalization
  - Semantic analysis
- Query Optimization
- Query Evaluation (Execution)

The Need for Optimization

Consider:

```
SELECT name, address
FROM Customer, Account
WHERE Customer.name = Account.name
      AND Balance > 2000
```

There are different possibilities for execution:

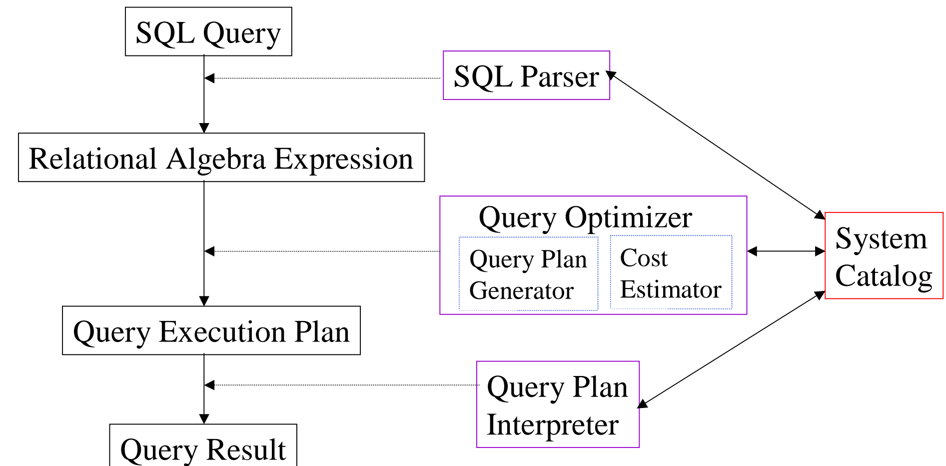
$\pi_{C.name,C.address}(\sigma_{C.name=A.name \wedge A.balance>2000}(C \times A))$

$\pi_{C.name,C.address}(\sigma_{C.name=A.name}(C \times \sigma_{A.balance>2000}(A))$

General Approaches to Optimization

- Heuristic-based query optimization
  - Given a query expression, perform selections and projections as early as possible.
  - Eliminate duplicate computations.
- Cost-based query optimization
  - Estimate the cost of different equivalent query expressions (using the heuristics and algebra manipulation) and choose the execution plan with the lowest cost estimation.

Architecture for DBMS Query Processing



General Guidelines

- Perform Selections and projections as early as possible
  - Splitting selection formula if necessary
  - Adding projections to eliminate unused columns
- Eliminating or reducing if possible repeated computations
- Combine unary operators with binary operators

Heuristic Transformations

Selection and projection-based transformations

- Cascading Selection

$$\sigma_{cond1 \wedge cond2}(R) \equiv \sigma_{cond1}(\sigma_{cond2}(R))$$

- Commutativity of selection

$$\sigma_{cond1}(\sigma_{cond2}(R)) \equiv \sigma_{cond2}(\sigma_{cond1}(R))$$

- Cascading of Projection

$$\pi_{Attribs1}(\pi_{Attribs2}(\ldots(\pi_{Attribsn}(R)\ldots)) \equiv \pi_{Attribs1}(R)$$

- Commutativity of Selection and Projection

$$\pi_{Attribs}(\sigma_{cond}(R)) \equiv \sigma_{cond}(\pi_{Attribs}(R))$$

Heuristic Transformations

Pushing selections and projections through joins

$$\sigma_{cond}(R \times S) \equiv R \bowtie_{cond} S$$

if conditions cond relate to the attributes of both R and S

$$\sigma_{cond}(R \times S) \equiv \sigma_{cond}(R) \times S$$

if attributes in cond all belong to R (idem with joins)

$$\pi_{Attribs1}(R \times S) \equiv \pi_{Attribs1}(\pi_{Attribs2}(R) \times S)$$

Where attribs1 $\subseteq$ attribs2 $\subseteq$ (R)

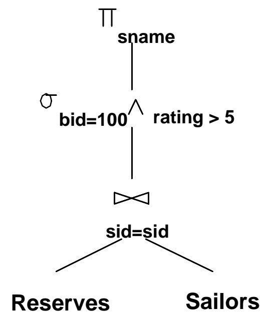
$$\pi_{Attribs1}(R \bowtie_{cond} S) \equiv \pi_{Attribs1}(\pi_{Attribs2}(R) \bowtie_{cond} S)$$

Attribs2 should contain all attributes in cond

Query Trees

- A query tree is a tree structure that corresponds to a relational algebra expression such that:
  - Each leaf node represents an input relation;
  - Each internal node represents a relation obtained by applying one relational operator to its child nodes
  - The root relation represents the answer to the query
- Two query trees are equivalent if their root relations are the same (query result)
- A query tree may have different execution plans
- Some query trees and plans are more efficient to execute than others.

Example of Query Tree



Overview of Query Optimization

- *Plan:* *Tree of Relational Algebra operators with choice of algorithms for each operation.*
  - Each operator typically implemented using a `pull' interface: when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them.
- Two main issues:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: Want to find best plan.
- Practically: Avoid worst plans!

Query Processing and Optimization

- Query Processing and Planning
- System catalog
- Evaluation of Relational Operations
- Cost Estimation and Plan Selection
- Physical Database Design Issues
- Database Tuning

System Catalog

- A Database system maintains information about every relation and view it contains.
- This information is stored in special relations called catalog relations or data dictionary
- The data in the data dictionary is extensively used for query optimization

System Catalog Information

- For each relation
  - Relation name, file name, file structure
  - Attribute name and type for all attributes
  - Index name for all indexes on the relation
  - Integrity constrains on the relation
- For each index
  - Index name and structure
  - Search key attributes
- For each view
  - View name and definition

Statistics Stored

- Cardinality *(Ntuples(R))*: number of tuples in each relation
- Size *(Npages(R))*: number of pages for each relation
- Index Cardinality *(Nkeys(I))*: number of distinct key values
- Index Size *(INPages(I))*: number of pages for each index
- Index Height *(IHeight(I))*: number of nonleaf levels for each tree index
- Index Range: number of minimum *(ILow(I))* and maximum *(IHigh(I))* present key values for each index
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field, or attribute weight, etc.) are sometimes stored.

Query Processing and Optimization

- Query Processing and Planning
- System catalog
- Evaluation of Relational Operations
- Cost Estimation and Plan Selection
- Physical Database Design Issues
- Database Tuning

Estimating the Result Size

- Typical optimizers estimate the size of the relation resulting from a relational operation.
- The result size estimation plays an important role in cost estimation because the output of an operation can be the input of another operation.
- In a SELECT-FROM-WHERE query, the size of the result is typically the product of the cardinality of the relations in the FROM clause, adjusted by the reduction effect by the conditions in the WHERE clause.

Reduction Factor

- Reduction effect depends upon the terms in the condition
- Column=Value ➔ reduction factor estimated by r ≈ 1/Nkeys(I). A better estimate is possible if histograms are available.
- Column1=Column2 ➔ reduction factor estimated by r ≈ 1/(MAX(Nkeys(I1),Nkeys(I2))
- Column > Value ➔ reduction factor is estimated by r ≈ (High(I)-Value)/(High(I)-Low(I))
- Column IN (list of Values) reduction factor is estimated by the factor for Column=Value for all values in the list.

Evaluating Relational Operators

- Selection (σ)
- Projection (π)
- Join (⋈)
- Is there more than one way to execute these operations? Can we take advantage of some factors such as indexes, ordering, etc.
- Other operators (difference, union, aggregation, group by, etc.)

Evaluating the Selection

- Size of result approximated as *size of R * reduction factor*.
- With no index, unsorted: Must essentially scan the whole relation; cost is M (#pages in R).
- With an index on selection attribute: Use index to find qualifying data entries, then retrieve corresponding data records. (Hash index useful only for equality selections.)
- Retrieval cost depends also upon clustering
- Complex conditions ➔ conjunctive normal form

Evaluating the Projection

- Projections can generate duplicate tuples after removing unnecessary attributes.
- Removing duplicates is difficult ➔ different approaches
- Projection based on sorting
  - Produce the set of tuples with desired attributes
  - Sort tuples with all remaining attributes
  - Scan sorted result comparing adjacent tuples
- Projection based on Hashing
  - Partition result with hash function (if enough buffers)
  - Eliminate duplicates in partitions

Evaluating the Join

- Simple Nested Loop Join
- Block Nested Loop Join
- Index Nested Loop Join
- Sort-Merge Join
- Hash Join

R⋈S is very Common → Must be carefully optimized. $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient

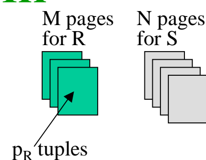
Schema for Examples

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.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Simple Nested Loops Join

```
foreach tuple r in R do
        foreach tuple s in S do
                if ri == sj  then add <r, s> to result
```

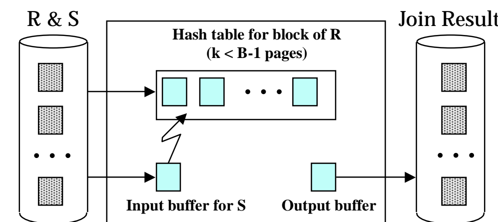


- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - Cost: $M + p_R * M * N$ = 1000 + 100*1000*500 I/Os. ≈(50 M)
- Page-oriented Nested Loops join: For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and s is in S-page.
  - Cost: M + M*N = 1000 + 1000*500 I/Os. ≈(501 10^3)
  - If smaller relation (S) is outer, cost = 500 + 500*1000 I/Os.

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
  - For each matching tuple r in R-blocks, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.

```
foreach block of B-2 of R do
   foreach page of S do
      forall matching in memory tuples
      r in R-blocks and s in S-Page
      add <r, s> to result
```



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil \# \text{ of pages of outer} / \text{blocksize} \rceil$
- With Reserves (R) as outer, and 100 pages of R:
  - Cost of scanning R is 1000 I/Os; a total of 10 (B-2) *blocks*.
  - ➔ we scan Sailors (S); 10*500 I/Os.
  - If space for just 90 pages of R, we would scan S 12 times ($\lceil 1000/90 \rceil$).
- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.

Index Nested Loops Join

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

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: $M + ((M*p_R)$ * cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples that match depends on clustering.
  - Clustered index: 1 I/O (typical since all matching tuples would be together), unclustered: up to 1 I/O per matching S tuple since they are scattered.

Examples of Index Nested Loops

- Hash-index on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 100,000 * 1.2 + 100,000 = 220,000 I/Os.
- Hash-index on *sid* of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.

Sort-Merge Join $(R \bowtie_{i=j} S)$

- Sort R and S on the join column, then scan them to do a \`\`merge'' (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

Example of Sort-Merge Join

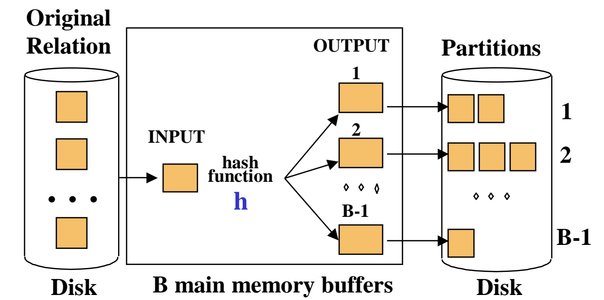
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

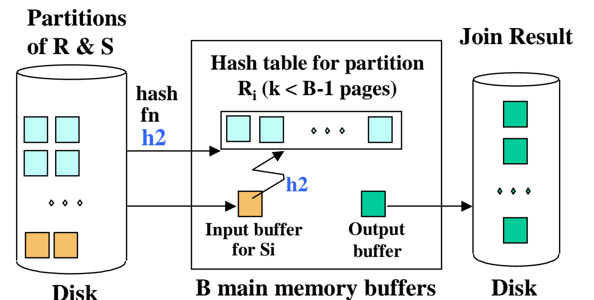
- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500 I/Os. However with BNL join could be less I/Os with 100 buffers

Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.



- Read in a partition of R, hash it using **h2** (≠ **h**). Scan matching partition of S, search for matches.
- Cost: Partitioning R/W once R and S= 2(M+N). Phase 2: read partitions once➔ M+N. Total 3(M+N)



Query Processing and Optimization

- Query Processing and Planning
- System catalog
- Evaluation of Relational Operations
- Cost Estimation and Plan Selection
- Physical Database Design Issues
- Database Tuning

Highlights of System R Optimizer

- Impact:
  - Most widely used currently; works well for < 10 joins.
- Cost estimation: Approximate art at best.
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
- Plan Space: Too large, must be pruned.
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be *pipelined* into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.

Schema for Examples

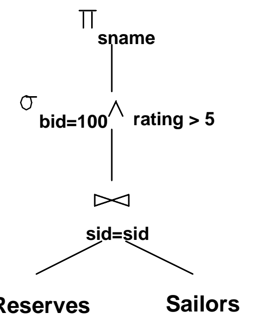
Sailors (<u>*sid*: integer</u>, *sname*: string, *rating*: integer, *age*: real)
Reserves (<u>*sid*: integer, *bid*: integer, *day*: dates</u>, *rname*: string)

- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

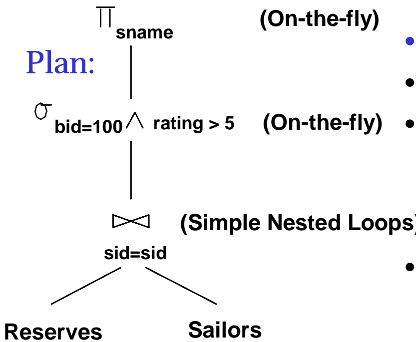
Motivating Example

```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
  R.bid=100 AND S.rating>5
```

RA Tree:

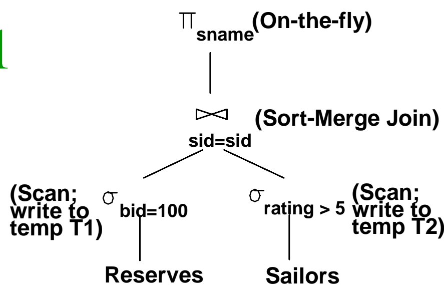


Plan:



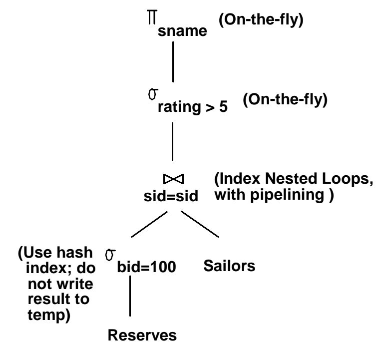
- Cost: 500+500*1000 I/Os
- By no means the worst plan!
- Misses several opportunities: selections could have been `pushed' earlier, no use is made of any available indexes, etc.
- *Goal of optimization:* Find more efficient plans that compute the same answer.

Alternative Plans 1 (No Indexes)



- ***Main difference:*** *<u>push selects.</u>*
- <u>With 5 buffers</u>, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
  - Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250)
  - Total: 4060 page I/Os.
- If we used BNL join, join cost = 10+4*250, total cost = 2770.
- If we `push' projections, T1 has only *sid*, T2 only *sid* and *sname*:
  - T1 fits in 3 pages, cost of BNL drops to under 250 pages, total < 2000.

Alternative Plans 2 With Indexes



- With clustered index on *bid* of Reserves, (100 boats) we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages for each boat.
- INL with ***<u>pipelining</u>*** (outer is not materialized).
  - –Projecting out unnecessary fields from outer doesn't help.
- Join column *sid* is a key for Sailors.
  - –At most one matching tuple, unclustered index on *sid* OK.
- Decision not to push *rating>5* before the join is based on availability of *sid* index on Sailors.
- Cost: Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000*1.2); total 1210 I/Os.

Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
    - We've already discussed how to estimate the cost of operations (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.
- The System R cost estimation approach.
  - Very inexact, but works OK in practice.
  - More sophisticated techniques known now.
- Query plans estimated at run-time or estimated once and elected plan stored and revisited for re-evaluation.

Size Estimation and Reduction Factors

- Consider a query block:

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

- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.
- *Reduction factor (RF)* associated with each *term* reflects the impact of the *term* in reducing result size. *Result cardinality* = Max # tuples * product of all RF's.
  - Implicit assumption that *terms* are independent!
  - Term *col=value* has RF *1/NKeys(I),* given index I on *col*
  - Term *col1=col2* has RF *1/MAX(NKeys(I1), NKeys(I2))*
  - Term *col>value* has RF *(High(I)-value)/(High(I)-Low(I))*

Summary

- Query optimization is an important task in a relational DBMS.
- Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- Two parts to optimizing a query:
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.

Query Processing and Optimization

- Query Processing and Planning
- System catalog
- Evaluation of Relational Operations
- Cost Estimation and Plan Selection
- Physical Database Design Issues
- Database Tuning

Overview

- After ER design, schema refinement, and the definition of views, we have the *conceptual* and *external* schemas for our database.
- The next step is to choose indexes, make clustering decisions, and to refine the conceptual and external schemas (if necessary) to meet performance goals.
- We must begin by understanding the *workload*:
  - The most important queries and how often they arise.
  - The most important updates and how often they arise.
  - The desired performance for these queries and updates.

Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Decisions to Make

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash/tree? Dynamic/static? Dense/sparse?
- Should we make changes to the conceptual schema?
  - Consider alternative normalized schemas? (Remember, there are many choices in decomposing into BCNF, etc.)
  - Should we ``undo'' some decomposition steps and settle for a lower normal form? *(Denormalization.)*
  - Horizontal partitioning, replication, views ...

Choice of Indexes

- One approach: consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: 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.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries, although it can help on equality queries as well in the presence of duplicates.
- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

Issues in Index Selection (Contd.)

- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - If range selections are involved, order of attributes should be carefully chosen to match the range ordering.
  - Such indexes can sometimes enable index-only strategies for important queries. (no need to access the relation)
    - For index-only strategies, clustering is not important!
- When considering a join condition:
  - Hash index on inner is very good for Index Nested Loops.
    - Should be clustered if join column is not key for inner, and inner tuples need to be retrieved.
  - *Clustered* B+ tree on join column(s) good for Sort-Merge.

Example 1

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

- Hash index on *D.dname* supports 'Toy' selection.
  - Given this, index on D.dno is not needed. Nothing is gained by an index on D.dno since Dept tuples are retrieved with dname index
- Hash 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.

Example 2

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

- Clearly, Emp should be the outer relation.
  - Suggests that we build a hash 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 selections.
- As both examples indicate, our choice of indexes is guided by the plan(s) that we expect an optimizer to consider for a query. *Have to understand optimizers!*

Examples of Clustering

```
SELECT E.dno
FROM Emp E
WHERE E.age>40
```

- 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.hobby=Stamps
```


Clustering and Joins

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

- Clustering is especially important when accessing inner tuples in INL.
  - Should make index on *E.dno* clustered.
- Suppose that the WHERE clause is instead:

  WHERE E.hobby='Stamps AND E.dno=D.dno
  - If many employees collect stamps, Sort-Merge join may be worth considering. A *clustered* index on D.dno would help.
- *Summary*: Clustering is useful whenever many tuples are to be retrieved.

Multi-Attribute Index Keys

- To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.
  - Such indexes also called *composite* or *concatenated* indexes.
  - Choice of index key orthogonal to clustering etc.
- If condition is: 20<*age*<30 AND 3000<*sal*<5000:
  - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is: *age*=30 AND 3000<*sal*<5000:
  - Clustered <*age,sal*> index much better than <*sal,age*> index!
- Composite indexes are larger, updated more often.

Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

<E.dno>

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

<E.dno,E.eid> Tree index!

```
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> Tree index!

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

<E. age,E.sal> or *<E.sal, E.age> Tree!*

```
SELECT AVG(E.sal)
FROM Emp E
WHERE E.age=25 AND
 E.sal BETWEEN 3000 AND 5000
```

Summary

- Database design consists of several tasks: *requirements analysis, conceptual design, schema refinement, physical design* and *tuning*.
  - In general, have to go back and forth between these tasks to refine a database design, and decisions in one task can influence the choices in another task.
- Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.
  - What are the important queries and updates? What attributes/relations are involved?

Summary (Contd.)

- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision; only one index on a given relation can be clustered!
  - Order of fields in composite index key can be important.
- Static indexes may have to be periodically re-built.
- Statistics have to be periodically updated.

Query Processing and Optimization

- Query Processing and Planning
- System catalog
- Evaluation of Relational Operations
- Cost Estimation and Plan Selection
- Physical Database Design Issues
- Database Tuning

Tuning the Conceptual Schema

- The choice of conceptual schema should be guided by the workload, in addition to redundancy issues:
  - We may settle for a 3NF schema rather than BCNF.
  - Workload may influence the choice we make in decomposing a relation into 3NF or BCNF.
  - We may further decompose a BCNF schema!
  - We might *denormalize* (i.e., undo a decomposition step), or we might add fields to a relation.
  - We might consider *horizontal decompositions*.
- If such changes are made after a database is in use, called *schema evolution*; might want to mask some of these changes from applications by defining *views*.

Example Schemas

Contracts (Cid, Sid, Jid, Did, Pid, Qty, Val)
Depts (Did, Budget, Report)
Suppliers (Sid, Address)
Parts (Pid, Cost)
Projects (Jid, Mgr)

- We will concentrate on Contracts, denoted as CSJDPQV. The following ICs are given to hold: JP $\rightarrow$ C, SD $\rightarrow$ P, C is the primary key.
  - What are the candidate keys for CSJDPQV?
  - What normal form is this relation schema in?

Settling for 3NF vs BCNF

- CSJDPQV can be decomposed into SDP and CSJDQV, and both relations are in BCNF. (Which FD suggests that we do this?)
  - Lossless decomposition, but not dependency-preserving.
  - Adding CJP makes it dependency-preserving as well.
- Suppose that this query is very important:
  - *Find the number of copies Q of part P ordered in contract C.*
  - Requires a join on the decomposed schema, but can be answered by a scan of the original relation CSJDPQV.
  - Could lead us to settle for the 3NF schema CSJDPQV.

Denormalization

- Suppose that the following query is important:
  - *Is the value of a contract less than the budget of the department?*
- To speed up this query, we might add a field *budget* B to Contracts.
  - This introduces the FD D $\rightarrow$ B wrt Contracts.
  - Thus, Contracts is no longer in 3NF.
- We might choose to modify Contracts thus if the query is sufficiently important, and we cannot obtain adequate performance otherwise (i.e., by adding indexes or by choosing an alternative 3NF schema.)

Choice of Decompositions

- There are 2 ways to decompose CSJDPQV into BCNF:
  - SDP and CSJDQV; lossless-join but not dep-preserving.
  - SDP, CSJDQV and CJP; dep-preserving as well.
- The difference between these is really the cost of enforcing the FD JP $\rightarrow$ C.
  - 2nd decomposition: Index on JP on relation CJP.
  - 1st:

```
CREATE ASSERTION CheckDep
CHECK ( NOT EXISTS ( SELECT *
FROM PartInfo P, ContractInfo C
WHERE P.sid=C.sid AND P.did=C.did
GROUP BY C.jid, P.pid
HAVING COUNT (C.cid) > 1 ))
```

Choice of Decompositions (Contd.)

- The following ICs were given to hold:
  JP → C, SD → P, C is the primary key.
- Suppose that, in addition, a given supplier always charges the same price for a given part: SPQ → V.
- If we decide that we want to decompose CSJDPQV into BCNF, we now have a third choice:
  - Begin by decomposing it into SPQV and CSJDPQ.
  - Then, decompose CSJDPQ (not in 3NF) into SDP, CSJDQ.
  - This gives us the lossless-join decomp: SPQV, SDP, CSJDQ.
  - To preserve JP → C, we can add CJP, as before.
- Choice: { SPQV, SDP, CSJDQ } or { SDP, CSJDQV } ?

Decomposition of a BCNF Relation

- Suppose that we choose { SDP, CSJDQV }. This is in BCNF, and there is no reason to decompose further (assuming that all known ICs are FDs).
- However, suppose that these queries are important:
  - *Find the contracts held by supplier S.*
  - *Find the contracts that department D is involved in.*
- Decomposing CSJDQV further into CS, CD and CJQV could speed up these queries. (Why?)
- On the other hand, the following query is slower:
  - *Find the total value of all contracts held by supplier S.*

Horizontal Decompositions

- Our definition of decomposition: Relation is replaced by a collection of relations that are *projections*. Most important case.
- Sometimes, might want to replace relation by a collection of relations that are *selections.*
  - Each new relation has same schema as the original, but a subset of the rows.
  - Collectively, new relations contain all rows of the original. Typically, the new relations are disjoint.

Horizontal Decompositions (Contd.)

- Suppose that contracts with value > 10000 are subject to different rules. This means that queries on Contracts will often contain the condition *val>10000*.
- One way to deal with this is to build a clustered B+ tree index on the *val* field of Contracts.
- A second approach is to 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
```

- The replacement of Contracts by LargeContracts and SmallContracts can be masked by the view.
- However, queries with the condition *val>10000* must be asked wrt LargeContracts for efficient execution: so users concerned with performance have to be aware of the change.

Tuning Queries and Views

- If a query runs slower than expected, check if an index needs to be re-built, 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.
  - Selections involving arithmetic or string expressions.
  - Selections involving OR conditions.
  - 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.

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:

```
SELECT  *  INTO  Temp
FROM  Emp E, Dept D
WHERE  E.dno=D.dno
        AND  D.mgrname='Joe'
```

and

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

vs.

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

Summary of Database Tuning

- The conceptual schema should be refined by considering performance criteria and workload:
  - May choose 3NF or lower normal form over BCNF.
  - May choose among alternative decompositions into BCNF (or 3NF) based upon the workload.
  - May *denormalize*, or undo some decompositions.
  - May decompose a BCNF relation further!
  - May choose a *horizontal decomposition* of a relation.
  - Importance of dependency-preservation based upon the dependency to be preserved, and the cost of the IC check.
    - Can add a relation to ensure dep-preservation (for 3NF, not BCNF!); or else, can check dependency using a join.

Summary (Contd.)

- Over time, indexes have to be fine-tuned (dropped, created, re-built, ...) for performance.
  - Should determine the plan used by the system, and adjust the choice of indexes appropriately.
- System may still not find a good plan:
  - Only left-deep plans considered!
  - Null values, arithmetic conditions, string expressions, the use of ORs, etc. can confuse an optimizer.
- So, may have to rewrite the query/view:
  - Avoid nested queries, temporary relations, complex conditions, and operations like DISTINCT and GROUP BY.