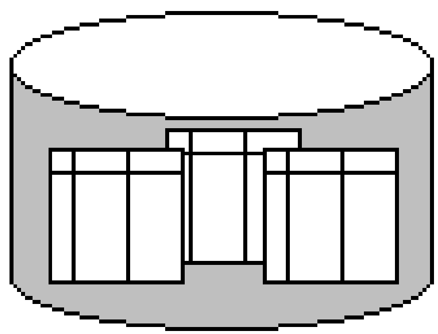
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Συστήματα Διαχείρισης Βάσεων Δεδομένων

Διάλεξη 4η: Physical and Logical Database Schema Tuning

Δημήτρης Πλεξουσάκης
Τμήμα Επιστήμης Υπολογιστών

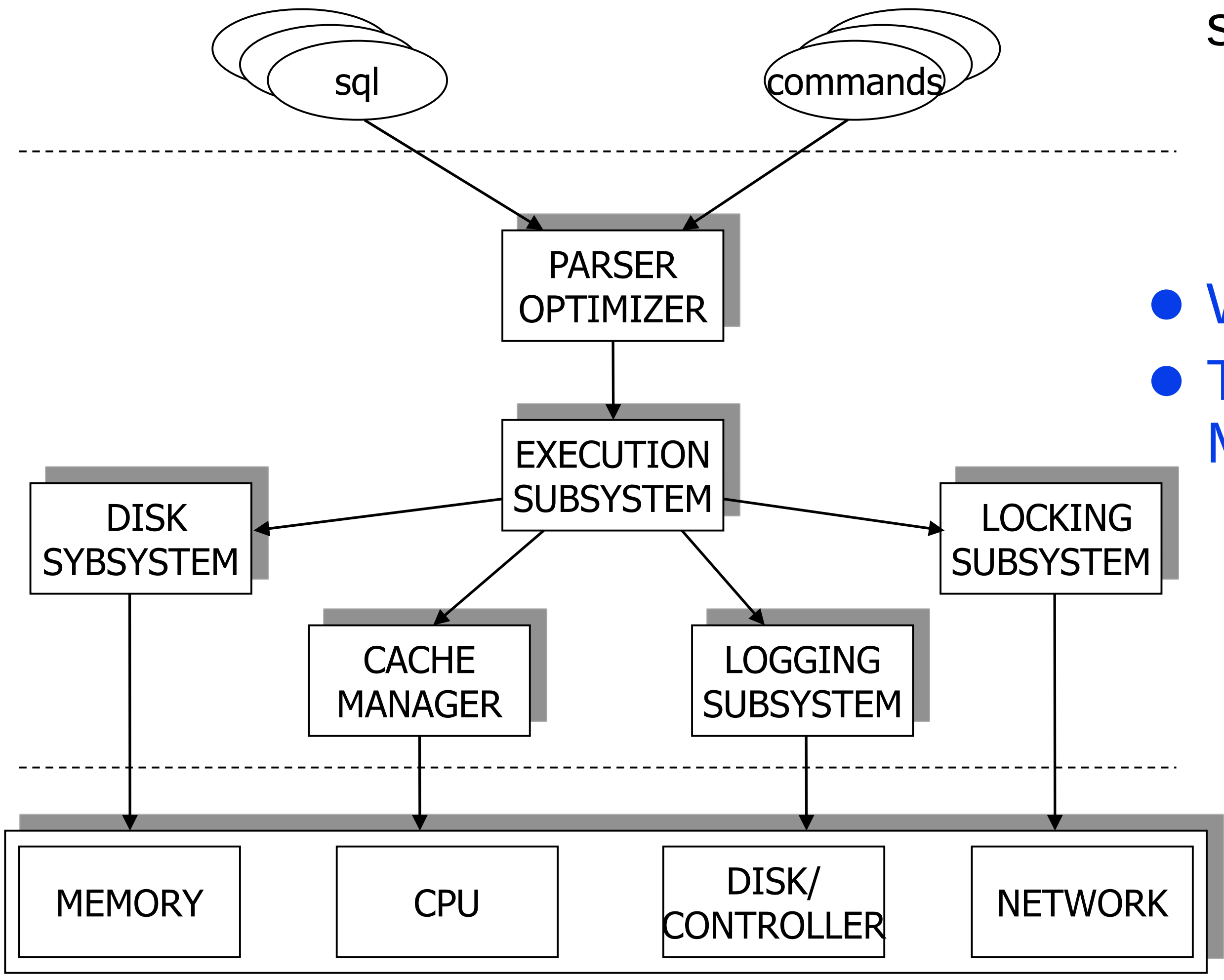
PHYSICAL AND LOGICAL DATABASE SCHEMA TUNING



Why Database Tuning?

- Troubleshooting (what is happening?):
  - Make managers and users happy given an application and a DBMS
- Capacity Sizing:
  - Buy the right DBMS given application requirements
- Application Programming:
  - Coding your application for performance

Why is Database Tuning hard?



The following query runs too slowly

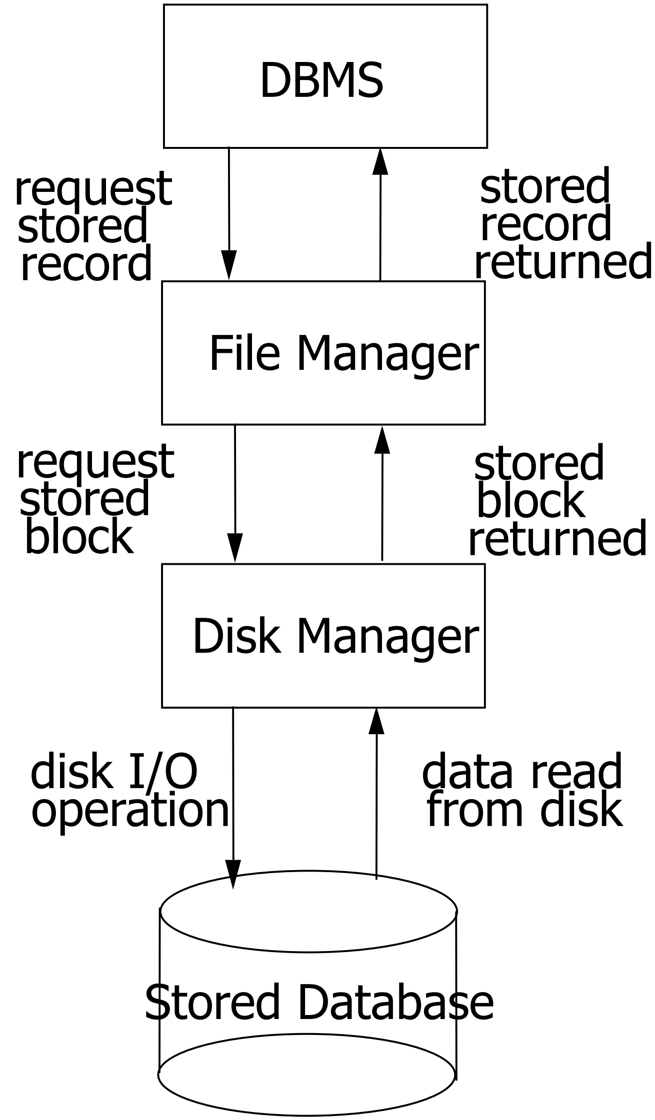
```
select *
from R
where R.a > 5;
```

- What do you do?
- Troubleshooting Methodology:
  - Hypothesis formulation
    - What is the cause of the problem?
  - Apply tuning principles to propose a fix
    - Hypothesis verification (experiments)

Tuning DB Design

- After designing schema
  - Make clustering decisions
  - Choose indexes
  - Refine the schemas (if necessary)

- We must begin by understanding the query 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 relations are going to be updated?
  - 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

Analyzing Database Queries and Transactions

- Expected frequency of invocation of queries and updates
  - expected frequency of each field as a selection field or join field over all transactions
  - expected frequency of retrieving and /or updating each record
- Analyzing time constraints of queries and updates
  - stringent performance constraints
  - influence access paths on selection fields
- Analyzing expected frequency of updates
  - volatile files
  - reduce number of access paths

Decisions to Make

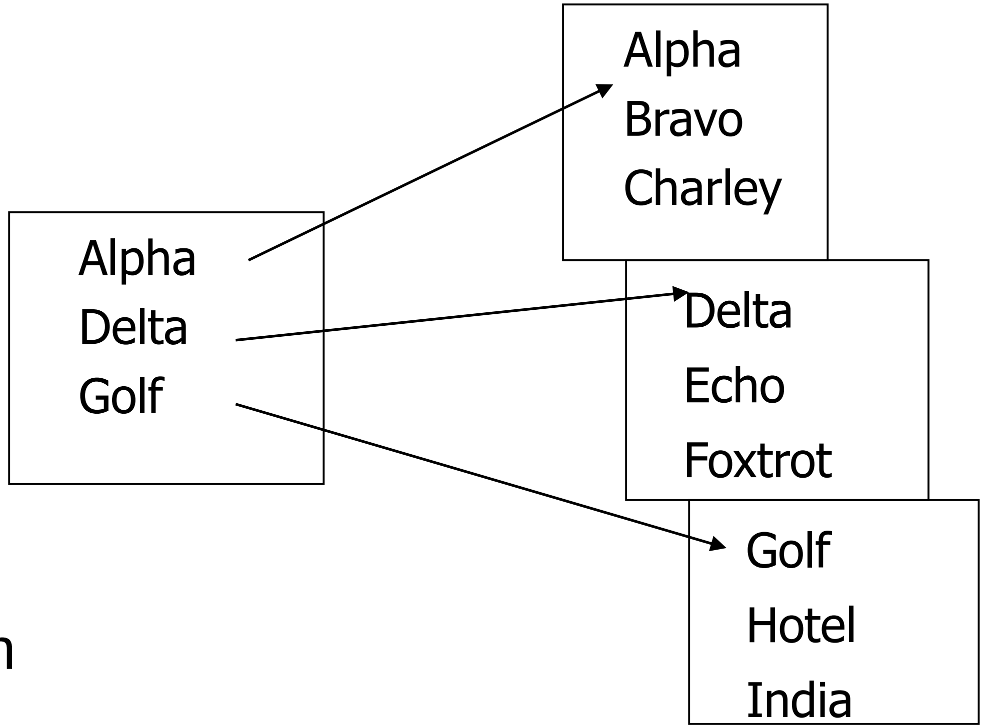
- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?

- For each index, what kind of index should it be?
  - Clustered? Hash/tree? Dynamic/static? Dense/sparse?

- Should we make changes to the schema?
  - Consider alternative normalized schemas? (Remember, there are many choices in decomposing into BCNF, etc.)
  - Horizontal partitioning, replication, views, ...

Recall Index Classification

- Index is a structure which provides alternative access to the data
  - Primary - key in index same as key in file
  - Secondary - key in index different from original file
  - Clustered - key order in index is same as data file (only one per table)
  - Unclustered - index tree stores sorted keys, with leaf node pointer to look up data (multiple per table)
  - Dense - one index entry for each record
  - Sparse - one index entry for each block
  - Covered - contain all columns in Select, Where, or Join clauses



Choice of Indexes

- One approach:
  - Consider the most important queries to tune
  - Consider the best plan using the current indexes
  - 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

- Create indexes on Primary Key columns (default clustered)
- Avoid indexes that are too wide
- Don't create indexes with less than 75% selectivity
  - Example: index on Yes/No column
- 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 to Consider in Index Selection

- 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

- When considering a join condition (indexes on foreign keys):
  - 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
- 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.mgr
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!

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>
dense unclusterred

```
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>
dense

```
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
```

Some Schemas are Better than Others

- A relation schema is a relation name and a set of attributes

  R(a int, b varchar[20]);

- A relation instance for R is a set of records over the attributes in the schema for R

- Schema1:

  ```
  OnOrder1(supplier_id,
    part_id, quantity,
    supplier_address)
  ```

- Schema 2:

  ```
  OnOrder2(supplier_id,
    part_id, quantity)
  Supplier(supplier_id,
    supplier_address)
  ```

- Space
  - Schema 2 saves space
- Information preservation
  - Some supplier addresses might get lost with schema 1
- Performance trade-off
  - Frequent access to address of supplier given an ordered part, then schema 1 is good
  - Many new orders, schema 1 is not good

Recall Functional Dependencies

- X is a set of attributes of relation R, and A is a single attribute of R: X determines A (the functional dependency X → A holds for R) iff:
  - For any relation instance I of R, whenever there are two records r and r' in I with the same X values, they have the same A value as well
    - `OnOrder1(supplier_id, part_id, quantity, supplier_address)`
      - `supplier_id → supplier_address` is an interesting FD
  - Attributes X from R constitute a key of R if X determines every attribute in R and no proper subset of X determines an attribute in R
    - `OnOrder1(supplier_id, part_id, quantity, supplier_address)`
      - `supplier_id, part_id` is not a key
    - `Supplier(supplier_id, supplier_address)`
      - `supplier_id` is a key

Recall Functional Dependencies

- A relation is normalized if every interesting functional dependency X → A involving attributes in R has the property that X is a key of R
  - `OnOrder1` is not normalized
  - `OnOrder2` and `Supplier` are normalized
- Relation R is in BCNF if: for any nontrivial FD X→Y of R, X must be a superkey
  - X→Y is nontrivial if Y is not a subset of X
  - X is a superkey if X→(all attributes of R)
  - Motivation: removing redundancy
- Relation R is in 3NF if: for each nontrivial FD X→Y, either X is a superkey, or Y is a member of some candidate key
  - Motivation: preserve FDs
- A BCNF relation is also a 3NF relation, but not vice versa

Tuning a Relational Schema

- The choice of relational 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, or add an attribute!
  - We might denormalize (i.e., undo a decomposition step)
  - 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

Tuning Normalization

- A single normalized relation XYZ is better than two normalized relations XY and XZ
  - if the single relation design allows queries to access X, Y and Z together without requiring a join
- The two-relation design is better iff:
  - Users access tend to partition between the two sets Y and Z most of the time
  - Attributes Y or Z have large values

Schema Tuning

- Rule of Thumb:
  - If ABC is normalized, and AB and AC are also normalized, then use ABC, unless:
    - Queries very rarely access ABC, but AB or AC (80% of the time)
    - Attribute B or C values are large
- Example
  - Schema 1:
    - `R1(bond_ID, issue_date, maturity, …)`
    - `R2(bond_ID, date, price)`
  - Schema 2:
    - `R1(bond_ID, issue_date, maturity, today_price, yesterday_price,…,10dayago_price)`

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 dependencies hold: JP → C, SD → P
  - C is the primary key
  - What are the candidate keys for CSJDPQV? C, JSD, JP
  - What normal form is this relation schema in? 3NF

Denormalization

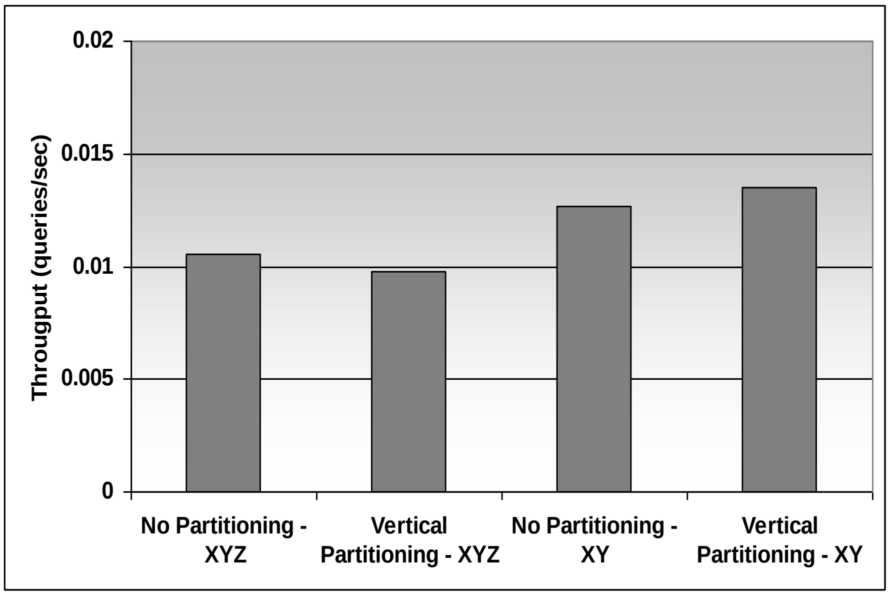
- Denormalizing means violating normalization for the sake of performance:
  - speeds up performance when attributes from different normalized relations are often accessed together
  - hurts performance for relations that are often updated
- Suppose that the following query Q is important:
  - Is the value of a contract less than the budget of the department?
  - Need a join between Contracts and Depts
- To speed up Q, we might add a field budget B to Contracts
  - This introduces the FD: D → B wrt Contracts
  - Thus, Contracts is no longer in 3NF
- We might choose to modify Contracts
  - 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)

(Vertical) Decomposition of a BCNF Relation

FD's: JP → C, SD → P Keys: C, JSD, JP

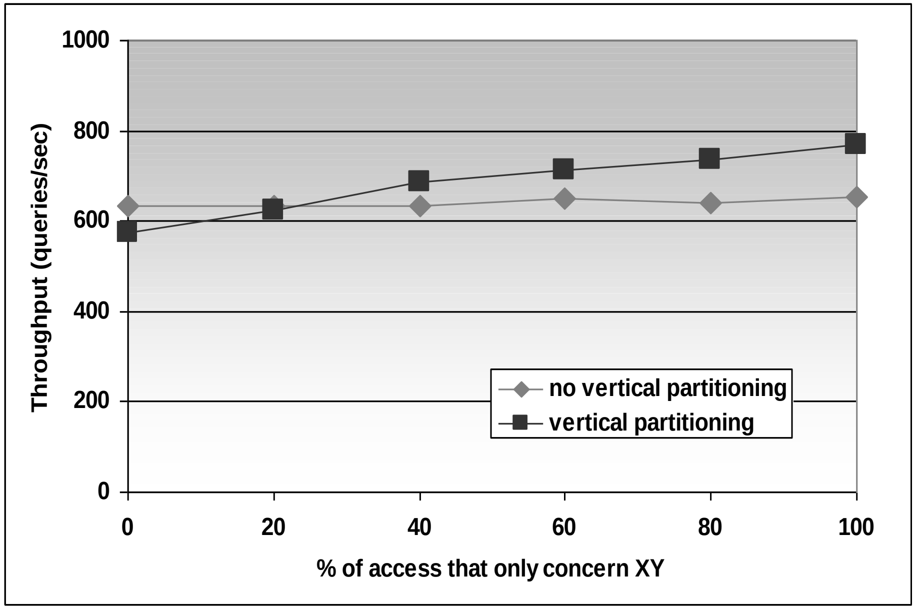
- Suppose we choose {SDP, CSJDQV}
  - Both are in BCNF
  - No reason to decompose further
- However, suppose that these queries are important:
  - Find the contracts held by supplier S
  - Find the contracts that department D is involved in
- Decomposing CSJDQV further into CS, CD and CJQV could speed up these queries (Why?)
- On the other hand, the following query is slower:
  - Find the total value of all contracts held by supplier S
  - Reason: need a join operation

Vertical Partitioning and Scan



- R (X,Y,Z)
  - X is an integer
  - YZ are large strings
- Scan Query
- Vertical partitioning exhibits poor performance when all attributes are accessed
- Vertical partitioning provides a sped up if only two of the attributes are accessed

Vertical Partitioning and Point Queries



- R (X,Y,Z)
  - X is an integer
  - YZ are large strings
- A mix of point queries access either XYZ or XY
- Vertical partitioning gives a performance advantage if the proportion of queries accessing only XY is greater than 20%
- The join is not expensive compared to a simple look-up

Horizontal Decompositions

- "Vertical" Decomposition: Relation is replaced by a collection of relations that are projections
- "Horizontal" decomposition
  - 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
- Suppose contracts with value > 10000 are very often
  - Queries on Contracts will often contain the condition val > 10000
- One 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

Denormalizing -- data

Settings:

lineitem (<u>L_ORDERKEY, L_PARTKEY , L_SUPPKEY</u>, L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE , L_DISCOUNT, L_TAX , L_RETURNFLAG, L_LINESTATUS , L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE, L_SHIPINSTRUCT , L_SHIPMODE , L_COMMENT);

region(<u>R_REGIONKEY</u>, R_NAME, R_COMMENT);

nation(<u>N_NATIONKEY</u>, N_NAME, *N_REGIONKEY*, N_COMMENT);

supplier(<u>S_SUPPKEY</u>, S_NAME, S_ADDRESS, *S_NATIONKEY*, S_PHONE, S_ACCTBAL, S_COMMENT);

- 600000 rows in lineitem, 25 nations, 5 regions, 500 suppliers

Denormalizing -- transactions

```
lineitemdenormalized (L_ORDERKEY, L_PARTKEY ,
    L_SUPPKEY, L_LINENUMBER, L_QUANTITY,
    L_EXTENDEDPRICE ,
    L_DISCOUNT, L_TAX , L_RETURNFLAG, L_LINESTATUS ,
    L_SHIPDATE, L_COMMITDATE,
    L_RECEIPTDATE, L_SHIPINSTRUCT ,
    L_SHIPMODE , L_COMMENT, L_REGIONNAME);
```

- 600000 rows in line item denormalized
- Cold Buffer
- Dual Pentium II (450MHz, 512Kb), 512 Mb RAM, 3x18Gb drives (10000RPM), Windows 2000

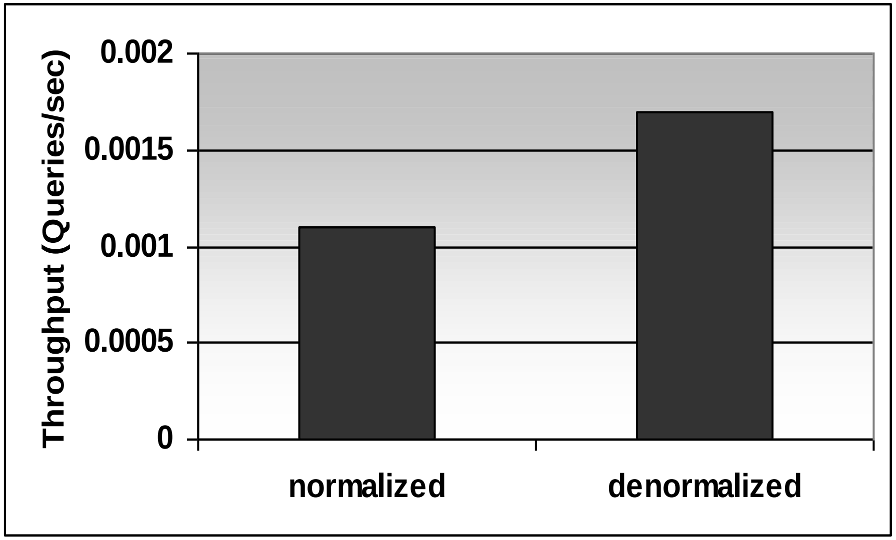
Queries on Normalized vs. Denormalized Schemas

Queries:

```
select L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER,
  L_QUANTITY, L_EXTENDEDPRICE, L_DISCOUNT, L_TAX,
  L_RETURNFLAG, L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
  L_RECEIPTDATE, L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT, R_NAME
from LINEITEM, REGION, SUPPLIER, NATION
where L_SUPPKEY = S_SUPPKEY
      and S_NATIONKEY = N_NATIONKEY
      and N_REGIONKEY = R_REGIONKEY
      and R_NAME = 'EUROPE';

select L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER,
  L_QUANTITY, L_EXTENDEDPRICE,  L_DISCOUNT, L_TAX,
  L_RETURNFLAG, L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
  L_RECEIPTDATE, L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT,
  L_REGIONNAME
from LINEITEMDENORMALIZED
where L_REGIONNAME = 'EUROPE';
```

Denormalization



- TPC-H schema
- Query: find all lineitems whose supplier is in Europe
- With a normalized schema this query is a 4-way join
- If we denormalize lineitem and add the name of the region for each lineitem (foreign key denormalization) throughput improves 30%

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 later in the this course…

Summary

- DB 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 DB 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
- 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

Summary

- Static indexes may have to be periodically re-built
- Statistics have to be periodically updated
- 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:
  - So, may have to rewrite the query/view
  - Avoid nested queries, temporary relations, complex conditions, and operations like DISTINCT and GROUP BY (more in following assisting lectures)

References

- Raghu Ramakrishnan and Johannes Gehrke - Database Management Systems, 3rd edition, McGraw-Hill 2002, chapter 16
- Dennis Shasha and Phillipe Bonnet - Tuning: Principles Experiments and Troubleshooting Techniques, Morgan Kaufmann Publishers 2002
- S, Chaudhuri, V. Narasayya An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server VLDB, Athens, 1997

Τέλος Ενότητας

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης
ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Σημειώματα

Σημείωμα αδειοδότησης

Σημείωμα Αναφοράς

Copyright Πανεπιστήμιο Κρήτης, Δημήτρης Πλεξουσάκης. «**Συστήματα Διαχείρισης Βάσεων Δεδομένων. Διάλεξη 4η: Physical and Logical Database Schema Tuning**». Έκδοση: 1.0. Ηράκλειο/Ρέθυμνο 2015. Διαθέσιμο από τη δικτυακή διεύθυνση: http://www.csd.uoc.gr/~hy460/