Data Administration in Information Systems

Database tuning

Database tuning

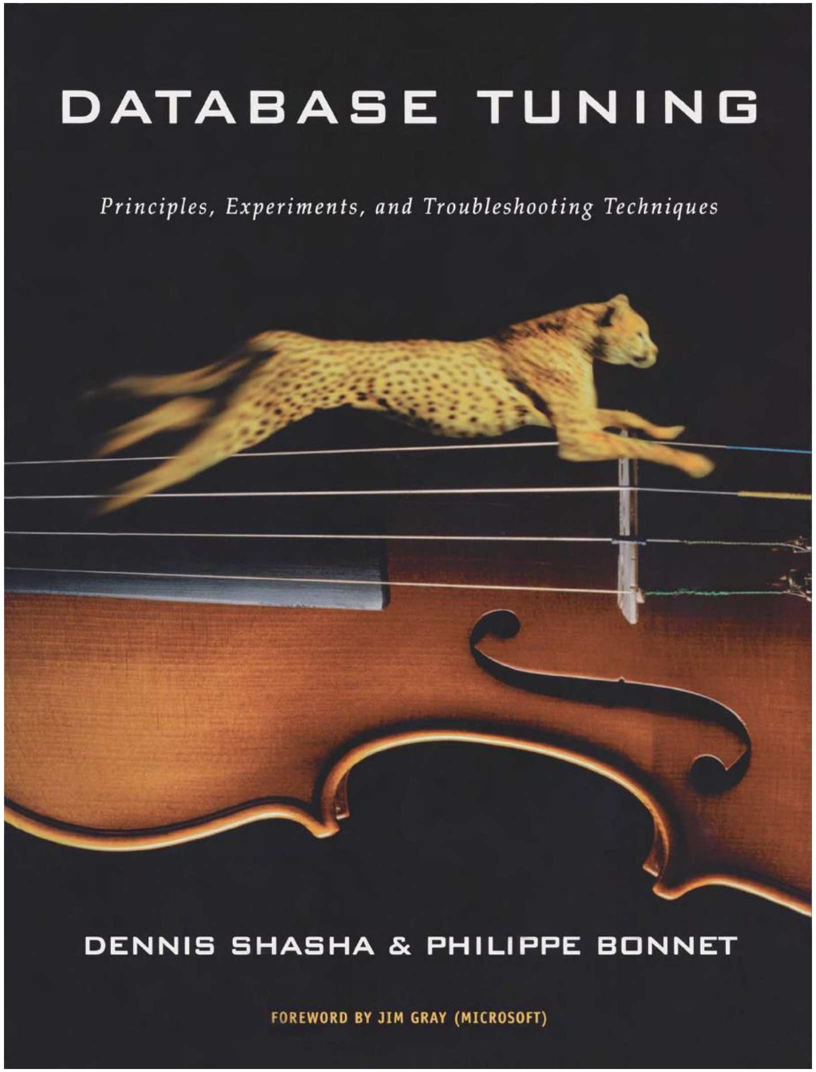


Table of Contents

What is Database Tuning?

- Activity of making a database application run faster
  - Faster = higher throughput, or lower response time
  - Avoiding transactions that create bottlenecks, or queries that run for hours unnecessarily, is a must
  - A 5% improvement is significant

Why Database Tuning?

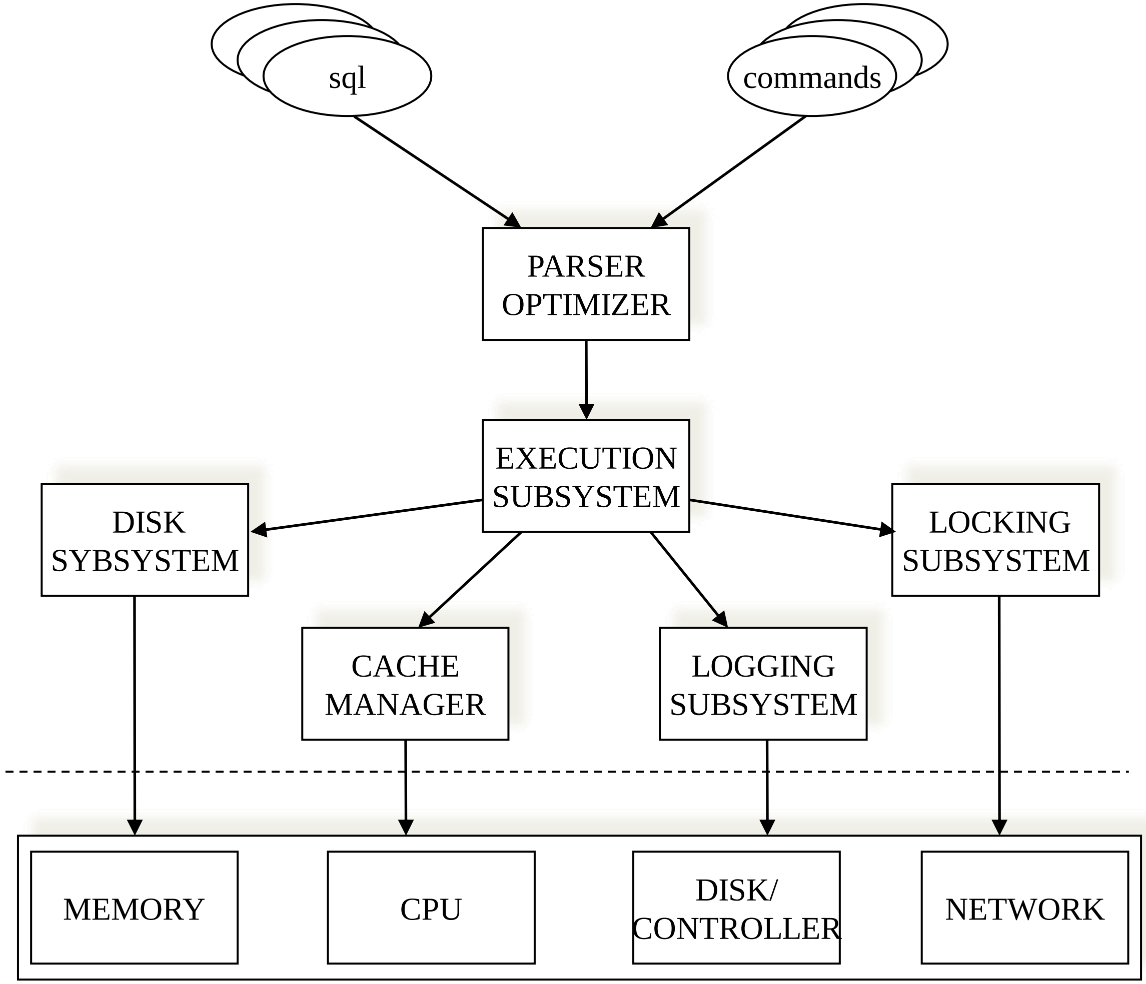
- Troubleshooting
  - Help managers and users overcome difficulties with a given application and database system
- Capacity Sizing
  - Help determine the right database system and hardware resources for given application requirements
- Application Programming
  - Help developers code their applications for performance

Why is Database Tuning hard?

- The following query runs too slowly

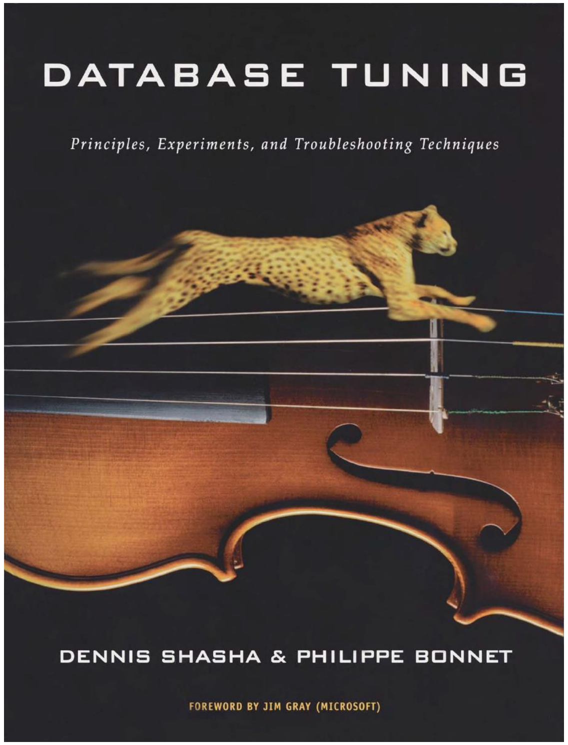
  select *
  from R
  where R.a > 5;

- But why??...



Database Tuning

- Second part for the course will address different tuning aspects, building on previous knowledge
  - Schema tuning
  - Query tuning
  - Index tuning
  - Lock and log tuning
  - Hardware and OS tuning
  - Database monitoring



Tuning Principles

1. Think globally, fix locally
2. Partitioning breaks bottlenecks
   - temporal and spatial
3. Start-up costs are high; running costs are low
4. Render unto server what is due unto server
5. Be prepared for trade-offs

Think globally, fix locally

- Proper identification of problem; minimal intervention
- Understand the whole, including the application goals before taking a set of queries and find the indexes that speed them up
- Example:
  - High I/O, paging and processor utilization may be due to frequent table scans instead of using an index, or due to log sharing a disk with some frequently accessed data.
  - Creating an index, or moving data files across different disks, may be cheaper and more efficient than buying an extra hard drive.

Partitioning breaks bottlenecks

- Technique for reducing the load on a certain component of the system, either by dividing the load over more resources or by spreading the load over time
- Partitioning may not always solve bottleneck:
  - First, try to speed up the component
  - If it doesn't work, partition
- Example:
  - Lock and resource contention among few long and many short transactions that access the same data.
  - Solution 1: run long transactions when there is little online transaction activity (partitioning in time).
  - Solution 2: allow long transactions (if read-only) to apply to out-of-date data on a separate disk (partitioning in space).

Start-up costs are high; running costs are low

- Obtain the effect you want with the fewest possible start-ups
- Examples:
  - It is expensive to begin a read operation on a disk, but once it starts, disk can deliver data at high speed.
    - So, frequently scanned tables should be laid out consecutively on disk.
  - Cost of parsing, semantic analysis, and optimizing the execution plan for some queries is non-negligible.
    - So, often executed queries should be compiled into the plan cache.

Render unto server what is due unto server

- Important design question is the allocation of work between the database system (server) and the application program (client)
- Depends on:
  - Relative computing resources of client and server: if the server is overloaded, tasks should be off-loaded to the clients
  - When something can be done efficiently on the DB (e.g. table joins), do it there before bringing out the data to the application
  - When the database task interacts with the user, then the part that waits for user input should be performed outside a transaction

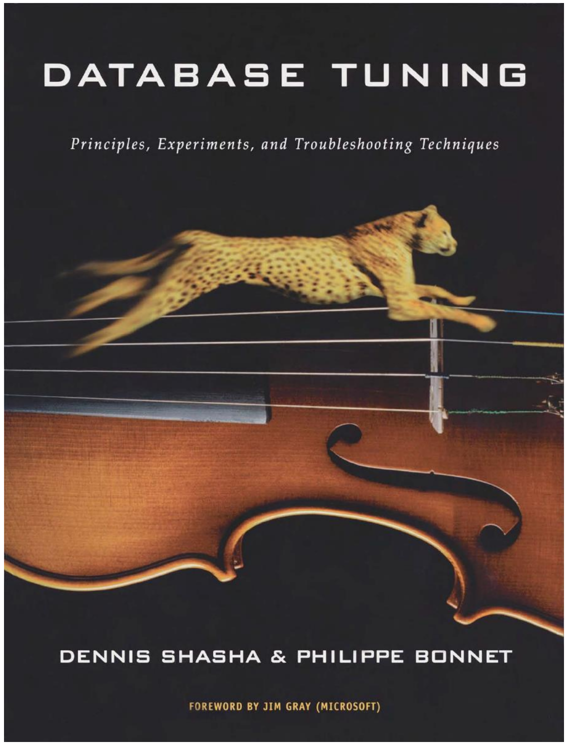
Be prepared for trade-offs

- Increasing speed of application requires combination of memory, disk and computational resources
- Examples:
  - Investing in RAM allows a system to increase its buffer size; this reduces the number of disk accesses and increases the system's speed.
  - Adding an index makes a critical query run faster, but requires more storage, more memory, and more disk accesses for insertions and updates.
  - When separating long queries from online updates, it would be nice to have a separate archival database for long queries; more performance, at the cost of purchasing and maintaining a separate computer system.

Database Tuning

- Second part for the course will address different tuning aspects, building on previous knowledge
  - Schema tuning
  - Query tuning
  - Index tuning
  - Lock and log tuning
  - Hardware and OS tuning
  - Database monitoring

Chapter 4



Tuning Schemas: Overview

- Trade-offs between normalization / de-normalization
  - Overview
  - When to normalize / de-normalize
- Vertical partitioning
  - Which queries benefit from partitioning
- Horizontal partitioning
- Aggregate maintenance and materialized views

Database Schemas (recap)

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

$$R(a \text{ int}, b \text{ varchar[20]}, \ldots)$$

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

NAME	ITEM	PRICE	QUANTITY	SUPPLIERNAME	YEAR
Bolt	4325	15	60	Standard Part	2001
Washer	5925	13	60	Standard Part	2002
Screw	6324	17	54	Standard Part	2003
Nut	3724	15	80	Metal Part	2001

Some Schemas Better than Others

- **Schema1 (unnormalized)**

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

- **Schema2 (normalized)**

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

100 000 orders

2 000 suppliers

supplier_id: 8-byte integer
supplier_address: 50 bytes

Some Schemas Better than Others

- **Schema1 (unnormalized)**

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

- **Schema2 (normalized)**

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

- Space
  - Schema 2 saves space, we are not repeating the *supplier_address*
- Update anomalies (information preservation)
  - Some supplier addresses might get lost with schema 1 if a supplier is deleted once the order has been filled
- Performance trade-off
  - In case of frequent accesses to supplier's address given an ordered part, then schema 1 is good, specially if there are few updates

Functional Dependency

- *X* is a set of attributes of relation *R*, and *A* is a single attribute of relation *R*.
- *X* determines *A*, i.e. functional dependency $X \rightarrow A$ holds for relation R, iff:
  - For any relation instance of *R*, whenever there are two records *r* and *r'* with the same *X* values, they have the same *A* value as well
  - This is *trivial* if *A* is part of *X*
  - It is *non-trivial* or *interesting* if *A* is not part of *X*

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

supplier_id → *supplier_address* is an interesting functional dependency

Key of a Relation

- Attributes *X* from *R* are a key of *R* if *X* determines every attribute in *R* and no proper subset of *X* determines an attribute in *R*
  - A key of a relation is a minimal set of attributes that determines all attributes in the relation

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

- (*supplier_id*, *part_id*) is a key
- *supplier_id* is not a key, because it does not determine *part_id*

Supplier(*supplier_id*, *supplier_address*)

- *supplier_id* is a key
- (*supplier_id*, *supplier_address*) is not a key, because it is not a minimal set of attributes that determines all attributes

Normalization

- A relation *R* is normalized if every interesting functional dependency $X \longrightarrow A$ has the property that *X* is a key of *R*

Schema1 (unnormalized)

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

Schema2 (normalized)

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

  – *OnOrder1* is not normalized, because the key is (*supplier_id*, *part_id*) but *supplier_id* alone determines *supplier_address*
  – *OnOrder2* and *Supplier* are normalized

Example 1

- Suppose that a bank associates each customer with his or her home branch. Each branch is in a specific legal jurisdiction.
  - Is the relation *R*(*customer*, *branch*, *jurisdiction*) normalized?
- What are the functional dependencies?
  - *customer* → *branch*
  - *branch* → *jurisdiction*
- The key is *customer*, but a functional dependency exists where *customer* is not involved.
  - ***R* is not normalized.**

Example 2

- Suppose that a doctor can work in several hospitals and receives a salary from each one.
  - Is *R*(*doctor*, *hospital*, *salary*) normalized?
- What are the functional dependencies?
  - *doctor*, *hospital* $\rightarrow$ *salary*
- The key is (*doctor*, *hospital*)
  - ***R* is normalized.**

Example 3

- Same relation *R*(*doctor*, *hospital*, *salary*) as before, but we add the doctor's primary home address.
  - Is *R*(*doctor*, *hospital*, *salary*, *primary_home_address*) normalized?
- What are the functional dependencies?
  - *doctor*, *hospital* → *salary*
  - *doctor* → *primary_home_address*
- Not normalized because *doctor* (a subset of the key) determines one attribute.
- A normalized decomposition would be:
  - *R1*(*doctor*, *hospital*, *salary*)
  - *R2*(*doctor*, *primary_home_address*)

Tuning Normalization

- Different normalization strategies may guide us to different sets of normalized relations
  – Which one to choose depends on the application's query patterns

Tuning Denormalization

- Denormalizing means **sacrificing normalization** for the sake of performance:
  - Denormalization **speeds up performance** when attributes from different normalized relations are often accessed together
  - Denormalization **hurts performance** for relations that are often updated

Denormalizing: Data

- Benchmark database

```
lineitem(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)
supplier(S_SUPPKEY, S_NAME, S_ADDRESS, S_NATIONKEY, S_PHONE,
         S_ACCTBAL, S_COMMENT)
nation(N_NATIONKEY, N_NAME, N_REGIONKEY, N_COMMENT)
region(R_REGIONKEY, R_NAME, R_COMMENT)
```

- 600 000 line items, 500 suppliers, 25 nations, 5 regions

Denormalizing: Denormalized Relation

- Query: find all line items whose supplier is in Europe

```
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)
```

- 600 000 line items

Queries on Normalized vs. Denormalized Schema

- Normalized:

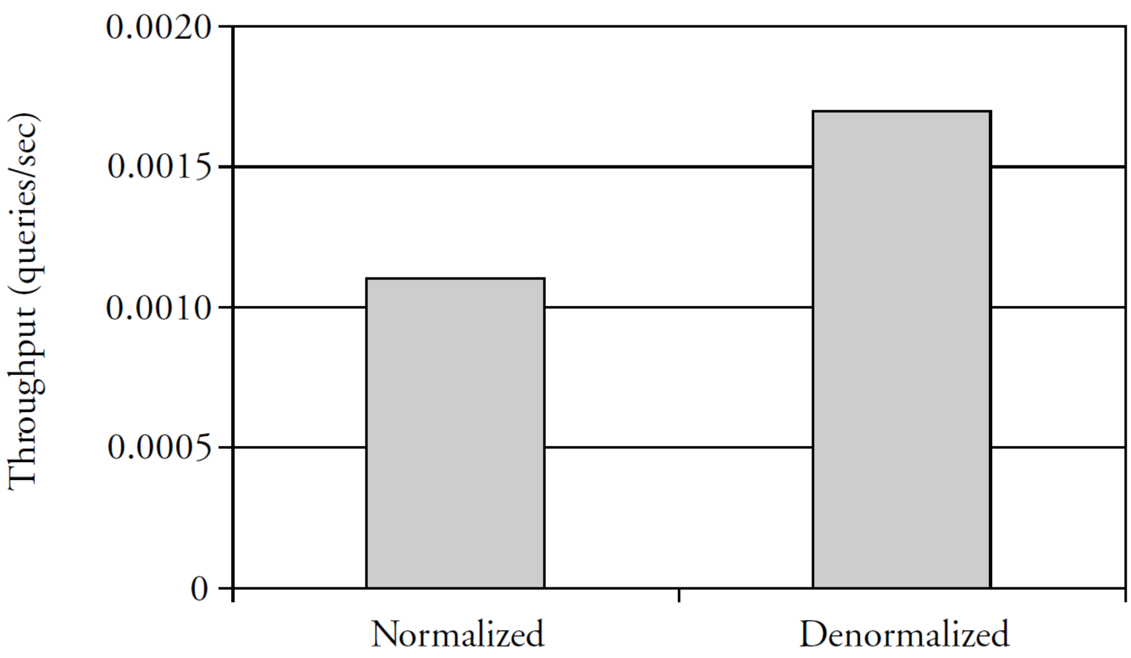
```
select L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER, L_QUANTITY,
       L_EXTENDEDPRICE, L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS,
       L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE, L_SHIPINSTRUCT,
       L_SHIPMODE, L_COMMENT, R_NAME
from LINEITEM, REGION, SUPPLIER, NATION
where L_SUPPKEY =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

- Denormalized:

```
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

- Benchmark database
  - Query: find all line items whose supplier is in Europe
- With a normalized schema this query is a **4-way join**
- If we denormalize and introduce the name of the region for each line item we obtain a **30% throughput improvement**



Vertical Partitioning: Example

- Three attributes: *account_ID*, *balance*, *address*
- Functional dependencies:
  - *account_ID* $\longrightarrow$ *balance*
  - *account_ID* $\longrightarrow$ *address*
- Two possible normalized schema designs:

  (*account_ID*, *balance*, *address*)

  or

  (*account_ID*, *balance*)

  (*account_ID*, *address*)

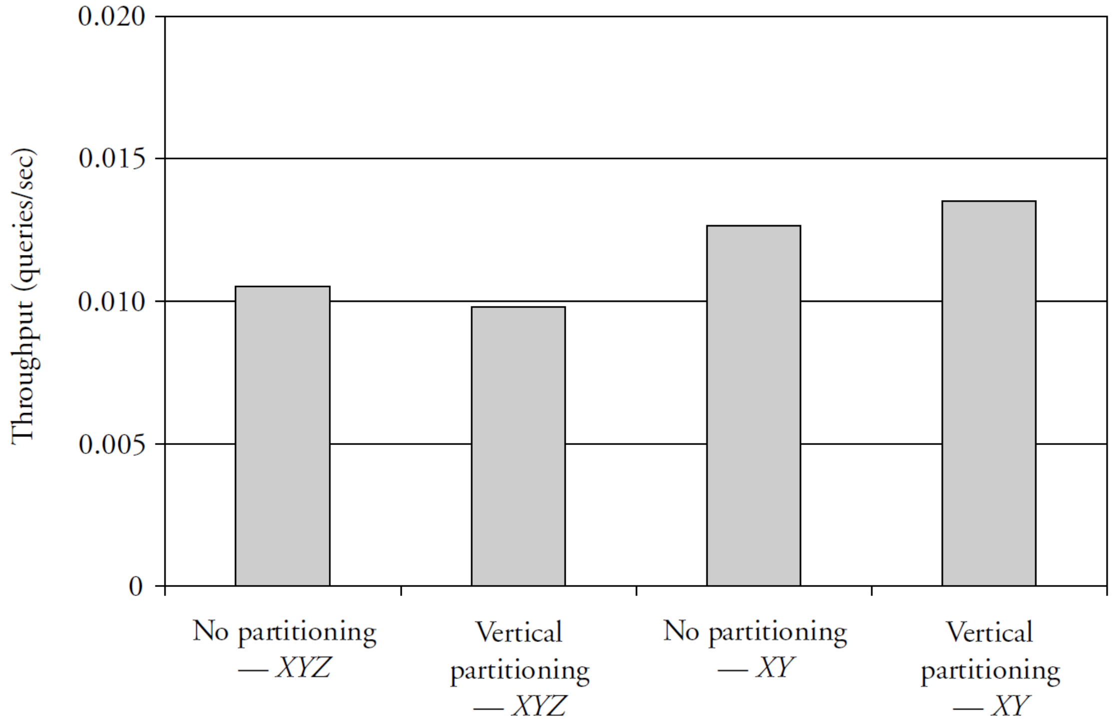
- Which design is better?

Vertical Partitioning

- It depends on the **query pattern**. Consider:
  - The address is used mainly by the application that sends a monthly account statement
  - The balance is updated or examined several times a day
- The second schema might be better because the relation (*account_ID*, *balance*) can be made **smaller**:
  - More (*account_ID*, *balance*) pairs fit in memory, thus increasing the hit ratio or cache efficiency
  - A scan performs better because there are fewer pages
- Here, two relations are better than one, even though they require more space

Vertical Partitioning

- *R*(<u>*X*</u>, *Y*, *Z*)
  - *X* is an integer key
  - *Y*, *Z* are large strings
- Vertical partitioning
  - R_1(<u>X</u>, Y)
  - R_2(<u>X</u>, Z)



- As expected:
  - Vertical partitioning exhibits **poor performance** when all attributes are accessed
  - Vertical partitioning provides a **speed up** if only two of the attributes are accessed

Vertical Partitioning: Rule

- A **single normalized relation XYZ** is better than two normalized relations XY and XZ for queries accessing X, Y, Z together
  - Those queries can access the three attributes without requiring a join
- The **two-relation design** is better if:
  - Accesses to X, Y and X, Z are separate most of the time
  - Attributes Y or Z have large values

Vertical Antipartitioning: Example

- A financial market database holds the closing price for the last 3000 trading days, however the 10 most recent trading days are especially important.

  (bond_id, issue_date, maturity, ...)

  (bond_id, date, price)

  vs.

  (bond_id, issue_date, maturity, price_today, price_yesterday, ...
  ..., price_10daysago)

  (bond_id, date, price)

- Second schema stores redundant info, requires extra space
  - Better for queries that need info about prices in the last 10 days, because it avoids a joining with thousands of dates

Horizontal Partitioning

- Until now, we have seen examples of **vertical partitioning**
  - Relation replaced by a collection of relations that are ***projections*** of the original schema
- Sometimes, it may instead be useful to partition a relation by a collection of relations that are ***selections***
  - Each new relation has the same schema, but a subset of the rows
  - Collectively, all relations contain all rows of the original relation
- Modern database systems can implement **horizontal partitioning** transparently to the user
  - E.g. partition schemes and partition functions in SQL Server

Horizontal Partitioning (Cont.)

- In a relation *R*(*ID, balance, address*), suppose that *accounts* with *balance* > 10000 are subject to different rules
  - Queries on *R* will often contain the condition balance > 10000

- One way to deal with this is to build a clustered B^+ tree index on the *balance* field of *R*

- A second approach is to replace *R* by two new relations, namely *LargeR* and *SmallR*, with the same attributes

- The replacement can be masked by a view involving a UNION of two selections, but queries with the condition value > 10000 must be asked to *LargeR*, for efficient execution

Aggregate Maintenance

- In reporting applications, aggregates (sums, averages, etc.) are often used
- For those queries it may be worthwhile to maintain special tables that hold those aggregates in pre-computed form
- Those tables are known as **materialized views**

Example

- The accounting department of a convenience store chain issues queries every 20 minutes to obtain:
  - The total dollar amount on order for a particular vendor
  - The total dollar amount on order by a particular store

- Original Schema:

  Orders(ordernum, itemnum, quantity, store, vendor)

  Item(itemnum, price)

  Store(store, name)

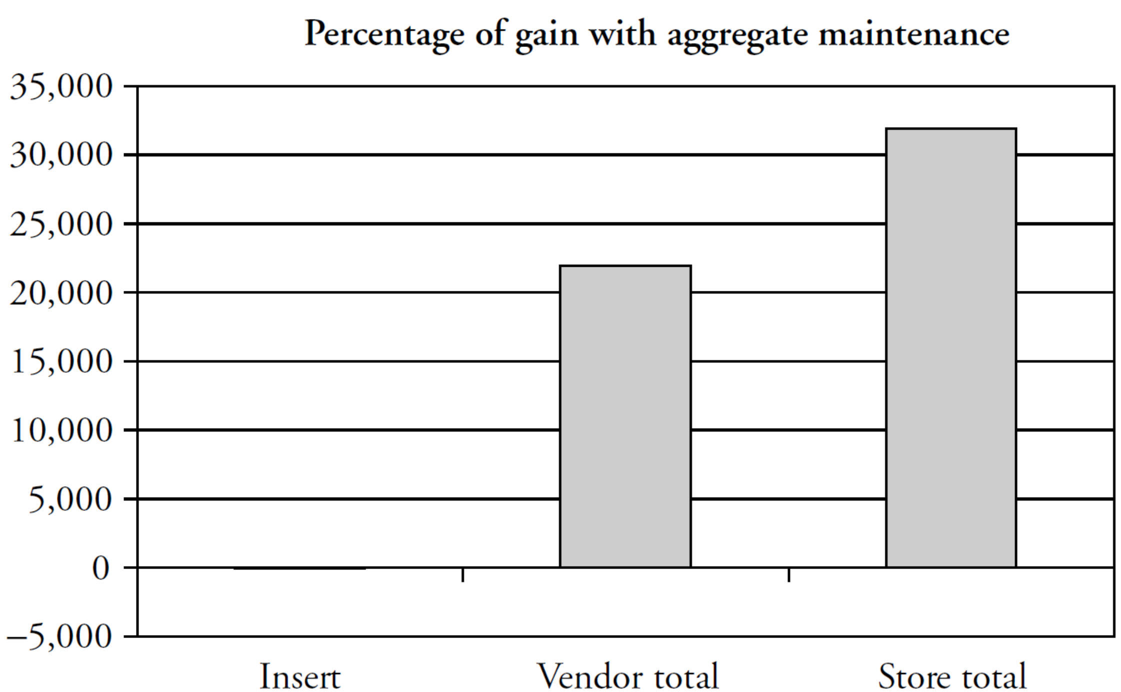
- The total dollar queries are expensive
  - vendor selection on Orders, join with Item on itemnum, multiply price*quantity, then sum
  - similarly for store, possibly requiring join with Store if selection by name

Solution: Aggregation Maintenance

- Add the following materialized views:
  - VendorTotal(vendor, amount), where amount is the dollar value of goods on order to the vendor, with a clustered index on vendor.
  - StoreTotal(store, amount), where amount is the dollar value of goods on order by the store, with a clustered index on store.
- Each update to Orders should update to these two views
  - materialized views take care of these updates implicitly
  - can also be implemented with tables updated by triggers

Aggregate Maintenance

- SQL Server on Windows
- 1 000 000 orders, 1 000 items
- Tables updated by triggers
- Insertions are 60% slower, queries are 20000% faster



Materialized Views in Oracle

- Oracle supports materialized views (so does Microsoft SQL Server – see labs):

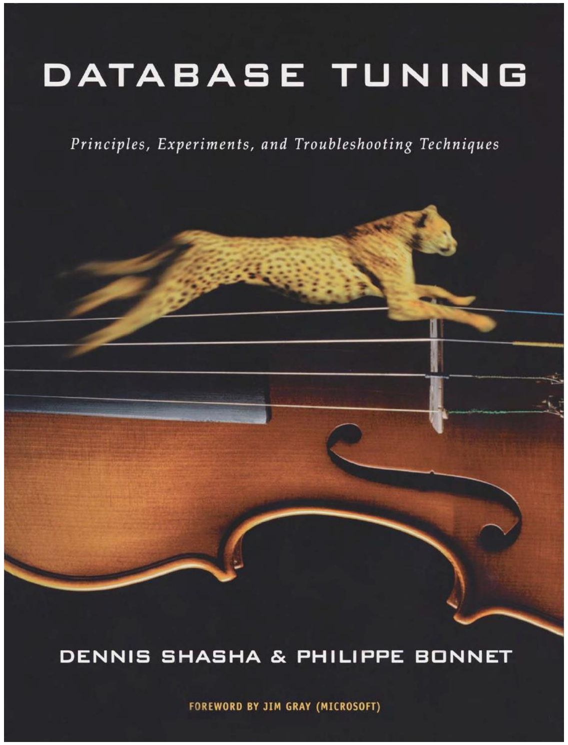
```
CREATE MATERIALIZED VIEW VendorOutstanding
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE
AS
SELECT orders.vendor, sum(orders.quantity*item.price)
FROM orders, item
WHERE orders.itemnum = item.itemnum
GROUP BY orders.vendor;
```

- Some Options:
  - BUILD immediate/deferred (when to populate the view)
  - REFRESH complete/incremental (how to keep the view updated)
  - ENABLE QUERY REWRITE (enable use for query optimization)
- Key characteristics:
  - Transparent aggregate maintenance
  - Transparent expansion performed by the optimizer based on cost
  - It is the optimizer and not the programmer that performs query rewriting

Database Tuning

- Second part for the course will address different tuning aspects, building on previous knowledge
  - Schema tuning
  - Query tuning
  - Index tuning
  - Lock and log tuning
  - Hardware and OS tuning
  - Database monitoring

Chapter 4



Query Tuning

```
SELECT s.RESTAURANT_NAME, t.TABLE_SEATING,
       to_char(t.DATE_TIME,'Dy, Mon FMDD') AS THEDATE,
       to_char(t.DATE_TIME,'HH:MI PM') AS THETIME,
       to_char(t.DISCOUNT,'99') || '%' AS AMOUNTVALUE,
       t.TABLE_ID, s.SUPPLIER_ID, t.DATE_TIME,
       to_number(to_char(t.DATE_TIME,'SSSSS')) AS SORTTIME
FROM TABLES_AVAILABLE t, SUPPLIER_INFO s,
     (SELECT s.SUPPLIER_ID, t.TABLE_SEATING, t.DATE_TIME,
             max(t.DISCOUNT) AMOUNT, t.OFFER_TYPE
      FROM TABLES_AVAILABLE t, SUPPLIER_INFO
      WHERE t.SUPPLIER_ID = s.SUPPLIER_ID
        and (TO_CHAR(t.DATE_TIME, 'MM/DD/YYYY') != TO_CHAR(sysdate, 'MM/DD/YYYY') OR
             TO_NUMBER(TO_CHAR(sysdate, 'SSSSS')) < s.NOTIFICATION_TIME - s.TZ_OFFSET)
        and t.NUM_OFFERS > 0
        and t.DATE_TIME > SYSDATE
        and s.CITY = 'SF'
        and t.TABLE_SEATING = '2'
        and t.DATE_TIME between sysdate and (sysdate + 7)
        and to_number(to_char(t.DATE_TIME, 'SSSSS')) between 39600 and 82800
        and t.OFFER_TYPE = 'Discount'
      GROUP BY s.SUPPLIER_ID, t.TABLE_SEATING, t.DATE_TIME, t.OFFER_TYPE) u
WHERE t.SUPPLIER_ID         = s.SUPPLIER_ID
  and u.SUPPLIER_ID = s.SUPPLIER_ID
  and t.SUPPLIER_ID = u.SUPPLIER_ID
  and t.TABLE_SEATING = u.TABLE_SEATING
  and t.DATE_TIME = u.DATE_TIME
  and t.DISCOUNT = u.AMOUNT
  and t.OFFER_TYPE = u.OFFER_TYPE
  and (TO_CHAR(t.DATE_TIME, 'MM/DD/YYYY') != TO_CHAR(sysdate, 'MM/DD/YYYY') OR
       TO_NUMBER(TO_CHAR(sysdate, 'SSSSS')) < s.NOTIFICATION_TIME - s.TZ_OFFSET)
  and t.NUM_OFFERS > 0
  and t.DATE_TIME > SYSDATE
  and s.CITY = 'SF'
  and t.TABLE_SEATING = '2'
  and t.DATE_TIME between sysdate and (sysdate + 7)
  and to_number(to_char(t.DATE_TIME, 'SSSSS')) between 39600 and 82800
  and t.OFFER_TYPE = 'Discount'
ORDER BY AMOUNTVALUE DESC, t.TABLE_SEATING ASC,
         upper(s.RESTAURANT_NAME) ASC,
         SORTTIME ASC, t.DATE_TIME ASC
```

- Execution is too slow…
  - How is this query executed?
  - How to make it run faster?

Query Monitoring

- Two ways to identify a slow query:
  - It issues far too **many disk accesses**, e.g., a query that scans an entire table
  - Its query plan, i.e. the plan chosen by the optimizer to execute the query, **fails to use promising indexes**

Query Rewriting

- The first tuning method to try is the one whose effects are purely local
  - Adding an index, changing the schema, modifying transactions have global effects that are potentially harmful
  - **Query rewriting** only impacts a particular query

Running Example

- **Employee**(ssnum, name, manager, dept, salary, numfriends)
  - Clustered index on *ssnum*
  - Non-clustered indexes (i) on *name* and (ii) on *dept*

- **Student**(ssnum, name, course, year)
  - Clustered index on *ssnum*
  - Non-clustered index on *name*

- **Techdept**(dept, manager, location)
  - Clustered index on *dept*

Query Rewriting Techniques

- Index usage
- Elimination of DISTINCT
- Nested queries
- Use of temporaries
- Join conditions
- Use of HAVING
- Use of views

Index Usage

- Many query optimizers **will not use indexes** in the presence of:
  - Arithmetic expressions
    ```
    WHERE salary/12 >= 4000;
    ```
  - Substring / upper / lower expressions
    ```
    SELECT * FROM Employee
    WHERE SUBSTR(name, 1, 1) = 'G';
    ```
  - Numerical comparisons of fields with different types
  - Comparison with NULL

Eliminate Unneeded DISTINCTs

- Query: Find employees who work in the information systems department. There should be no duplicates.

```
SELECT DISTINCT ssnum
FROM Employee
WHERE dept = 'Information Systems';
```

- **DISTINCT is unnecessary**, since *ssnum* is a key of employee so certainly is a key of a subset of employee.

Employee(ssnum, name, manager, dept, salary, numfriends)

Student(ssnum, name, course, year)

Techdept(dept, manager, location)

Eliminate Unneeded DISTINCTs (Cont.)

- Query: Find social security numbers of employees in tech departments. There should be no duplicates.

```
SELECT DISTINCT ssnum
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept;
```

- Is DISTINCT needed?

Employee(<u>ssnum</u>, name, manager, dept, salary, numfriends)

Student(<u>ssnum</u>, name, course, year)

Techdept(<u>dept</u>, manager, location)

DISTINCT Unnecessary

- Since *dept* is a key of the *Techdept* table, each employee record will join with at most one record in *Techdept*.
- So, each employee record will be part of at most one record of the join result.
- Because *ssnum* is a key for Employee, at most one record in *Employee* will have a given *ssnum* value, so DISTINCT is unnecessary.

```
SELECT DISTINCT ssnum
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept;
```

Employee(<u>ssnum</u>, name, manager, dept, salary, numfriends)

Student(<u>ssnum</u>, name, course, year)

Techdept(<u>dept</u>, manager, location)

When DISTINCT is Required

- In general, DISTINCT is required when:
  - The set of values or records returned should contain no duplicates
  - The columns returned do not contain a key of the relation created by the FROM and WHERE clauses

Reaching

- The relationship among DISTINCT, keys and joins can be generalized:
  - Call a table *T* **privileged** if the fields returned by the SELECT contain a key of *T*
  - Let *R* be an unprivileged table. Suppose that *R* is joined on equality by its key field to some other table *S*, then we say ***R* reaches *S***
  - Now, define *reaches* to be **transitive**. So, if *R1* reaches *R2* and *R2* reaches *R3* then say that *R1* reaches *R3*

Reaching: Main Theorem

- There will be no duplicates among the records returned by a selection, if one of the two following conditions hold:
  - Every table mentioned in the FROM clause is **privileged**
  - Every **unprivileged** table **reaches** at least one **privileged** table

Reaching: Proof Sketch

- If every relation is privileged then there are no duplicates
- Suppose some relation *T* is not privileged but reaches at least one privileged one, say *U*. Then the join clauses linking *T* with *U* ensure that each distinct combination of privileged records is joined with at most one record of *T*.

Reaching: Example 1

```
SELECT ssnum
FROM Employee, Techdept
WHERE Employee.manager = Techdept.manager;
```

- Returns duplicates
- The same Employee record may match several Techdept records (because *manager* is not a key of *Techdept*), so the *ssnum* of that employee record may appear several times
- The unprivileged relation *Techdept* does not reach the privileged relation *Employee*

Employee(ssnum, name, manager, dept, salary, numfriends)

Student(ssnum, name, course, year)

Techdept(dept, manager, location)

Reaching: Example 2

```
SELECT ssnum, Techdept.dept
FROM Employee, Techdept
WHERE Employee.manager = Techdept.manager;
```

- Does not return duplicates
- Each repetition of a given *ssnum* value would be accompanied by a new *Techdept.dept* since *Techdept.dept* is a key of *Techdept*
- Both relations are privileged

Employee(ssnum, name, manager, dept, salary, numfriends)

Student(ssnum, name, course, year)

Techdept(dept, manager, location)

Reaching: Example 3

```
SELECT Student.ssnum
FROM Student, Employee, Techdept
WHERE Student.ssnum = Employee.ssnum
  AND Employee.dept = Techdept.dept;
```

- Does not return duplicates
- *Student* is privileged
- Both *Employee* and *Techdept* reach *Student*

Employee(ssnum, name, manager, dept, salary, numfriends)

Student(ssnum, name, course, year)

Techdept(dept, manager, location)

Types of Nested Queries

- Uncorrelated subqueries with aggregates in the nested query

```
SELECT ssnum
FROM Employee
WHERE salary > (SELECT avg(salary)
                FROM Employee);
```

- Uncorrelated subqueries without aggregates in the nested query

```
SELECT ssnum
FROM Employee
WHERE dept IN (SELECT dept
               FROM Techdept);
```

Types of Nested Queries (Cont.)

- Correlated subqueries with aggregates

```
SELECT ssnum
FROM Employee e1
WHERE salary > (SELECT avg(e2.salary)
                FROM Employee e2, Techdept
                WHERE e2.dept = e1.dept
                  AND e2.dept = Techdept.dept);
```

- Correlated subqueries without aggregates

```
SELECT ssnum
FROM Employee e1
WHERE EXISTS (SELECT *
              FROM Techdept t1
              WHERE e1.dept = t1.dept);
```

Rewriting of Uncorrelated Subqueries

1. Retain the SELECT clause from the outer block
2. Combine the arguments of the two FROM clauses
3. AND together all the WHERE clauses, replacing IN by =

```
SELECT ssnum
FROM Employee
WHERE dept IN (SELECT dept
               FROM Techdept);
```

becomes

```
SELECT ssnum
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept;
```

Rewriting of Uncorrelated Subqueries

- Potential problem with duplicates

```
SELECT avg(salary)
FROM Employee
WHERE manager IN (SELECT manager
                  FROM Techdept);
```

rewritten as

```
SELECT avg(salary)
FROM Employee, Techdept
WHERE Employee.manager = Techdept.manager;
```

- The rewritten query may include an employee record several times if that employee's manager manages several departments.

Rewriting of Uncorrelated Subqueries

- Solution: use a temporary table with DISTINCT to eliminate duplicates from the nested relation

```
SELECT DISTINCT manager INTO Temp
FROM Techdept;

SELECT avg(salary)
FROM Employee, Temp
WHERE Employee.manager = Temp.manager;
```

Rewriting of Correlated Subqueries

- Query: find the employees who earn more than the average salary in their tech department

```
SELECT ssnum
FROM Employee e1
WHERE salary > (SELECT avg(e2.salary)
                FROM Employee e2, Techdept
                WHERE e2.dept = Techdept.dept
                  AND e2.dept = e1.dept);
```

- This could be inefficient; same average salary computed multiple times

Rewriting of Correlated Subqueries

- Solution

```
INSERT INTO Temp
SELECT avg(salary) as avsalary, Employee.dept
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
GROUP BY Employee.dept;
```

- Returns the average of salaries per tech department

```
SELECT ssnum
FROM Employee, Temp
WHERE salary > avsalary
  AND Employee.dept = Temp.dept;
```

Rewriting of Correlated Subqueries

- Possible problem
- Query: Find employees whose number of friends equals the number of employees in their tech department

```
SELECT ssnum
FROM Employee e1
WHERE numfriends = (SELECT COUNT(e2.ssnum)
                    FROM Employee e2, Techdept
                    WHERE e2.dept = Techdept.dept
                      AND e2.dept = e1.dept);
```

Employee(ssnum, name, manager, dept, salary, numfriends)

Student(ssnum, name, course, year)

Techdept(dept, manager, location)

Rewriting of Correlated Subqueries

- Solution would be...

```
INSERT INTO Temp
SELECT COUNT(ssnum) as numcolleagues, Employee.dept
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
GROUP BY Employee.dept;

SELECT ssnum
FROM Employee, Temp
WHERE numfriends = numcolleagues
  AND Employee.dept = Temp.dept;
```

Rewriting of Correlated Subqueries

- The COUNT bug
  - Let us consider Ana who is not in a tech department.
  - In the **original query**, Ana's number of friends would be compared to the count of an empty set, which is 0. In case Ana has no friends, she would survive the selection.
  - In the **transformed query**, Ana's record would not appear because she does not work for a tech department.
  - This is a limitation of the correlated subquery rewriting technique when COUNT is involved.

(Ab)use of Temporaries

- Query: Find all employees in the information systems department who earn more than $40000

```
INSERT INTO Temp
SELECT *
FROM Employee
WHERE salary > 40000;

SELECT ssnum
FROM Temp
WHERE Temp.dept = 'Information Systems';
```

- Optimizer would miss the opportunity to use the index on dept

(Ab)use of Temporaries

- More efficient solution

```sql
SELECT ssnum
FROM Employee
WHERE dept = 'Information Systems'
  AND salary > 40000;
```

Join Conditions

- It is a good idea to express join conditions on clustered indexes.
  - Possibility of using merge join without need for sorting

- If that fails, it is a good idea to express join conditions on numerical attributes rather than on string attributes

Join Conditions

- Example: Find all students who are also employees

```
SELECT *
FROM Employee, Student
WHERE Employee.name = Student.name;
```

- Both tables have index on *name*, but it is a non-clustered index; the following join would be much more efficient:

```
SELECT *
FROM Employee, Student
WHERE Employee.ssnum = Student.ssnum;
```

Use of HAVING

- Do not use HAVING when WHERE is enough

```
SELECT avg(salary) as avgsalary, dept
FROM Employee
GROUP BY dept
HAVING dept = 'Information Systems';
```

```
SELECT avg(salary) as avgsalary, dept
FROM Employee
WHERE dept = 'Information Systems'
GROUP BY dept;
```

Use of HAVING

- HAVING should be reserved for **aggregates** on groups

```
SELECT avg(salary) as avgsalary, dept
FROM Employee
GROUP BY dept
HAVING count(ssnum) > 100;
```

Use of Views

- Views may cause queries to execute inefficiently

```
CREATE VIEW Techlocation AS
SELECT ssnum, Techdept.dept, location
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept;
```

```
SELECT dept
FROM Techlocation
WHERE ssnum = 43253265;
```

- Optimizers expand views when identifying the query blocks to be optimized

Use of Views

- The selection from *Techlocation* is expanded into a join:

```
SELECT dept
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
AND ssnum = 43253265;
```

- But the following less expensive query is possible, since dept is an attribute of Employee

```
SELECT dept
FROM Employee
WHERE ssnum = 43253265;
```

Performance Impact of Query Rewritings