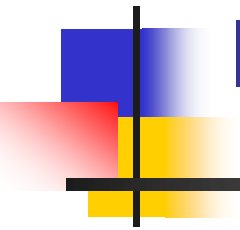
Relational Data Management & Data Management Fundamentals

Damianos Chatziantoniou (damianos@aueb.gr)
Department of Management Science and Technology
Athens University of Economics and Business


Outline

- Introduction
- Data Modeling
- Query Languages
- Query Processing
- Physical Implementation
- Transactions
- Parallel Databases
- Distributed Databases



Introduction

Using Programming Languages: Files

- 60s, 70s → data management through programming languages (e.g. COBOL, Fortran)
- An example to create and write to a text file: java file
- An example to read a text file: java file
- So… why do we need data management systems?


Need for Database Management Systems

- In the early days, database applications were built directly on top of file systems
- Drawbacks:
  - Data redundancy and inconsistency
    - Multiple file formats, duplication of information in different files
  - Difficulty in accessing data
    - Need to write a new program to carry out each new task
  - Data isolation — multiple files and formats
  - Security problems
    - Hard to provide user access to some, but not all, data

Need for Database Management Systems

- Drawbacks (cont.):
  - Integrity problems
    - Integrity constraints (e.g., account balance > 0) become "buried" in program code rather than being stated explicitly
    - Hard to add new constraints or change existing ones
  - Atomicity of updates
    - Failures may leave database in an inconsistent state with partial updates carried out
    - Example: Transfer of funds from one account to another should either complete or not happen at all
  - Concurrent access by multiple users
    - Concurrent access needed for performance
    - Uncontrolled concurrent accesses can lead to inconsistencies


Designing a Data Mgmt Application

- Real-world situation – Understand the problem
- Requirement Analysis - Interviews
- Modeling – Conceptual + Logical data modeling
- Queries – Query Languages, processing, execution
- Physical Implementation – Indexing, Storage


What is a Data Model?

- A set of constructs to describe
  - data
  - relationships between data
  - data semantics
  - constraints on data


What is a Query Language?

- There must be a way to retrieve data from the data management system. Options:
  - programs (e.g. C, Python, Java)
  - specialized languages (e.g. SQL, QBE)
- There is a strong coupling with the data model
  - Relational → SQL
  - XML → XPath, XQuery
  - Graph → Cypher


Evaluation Algorithms

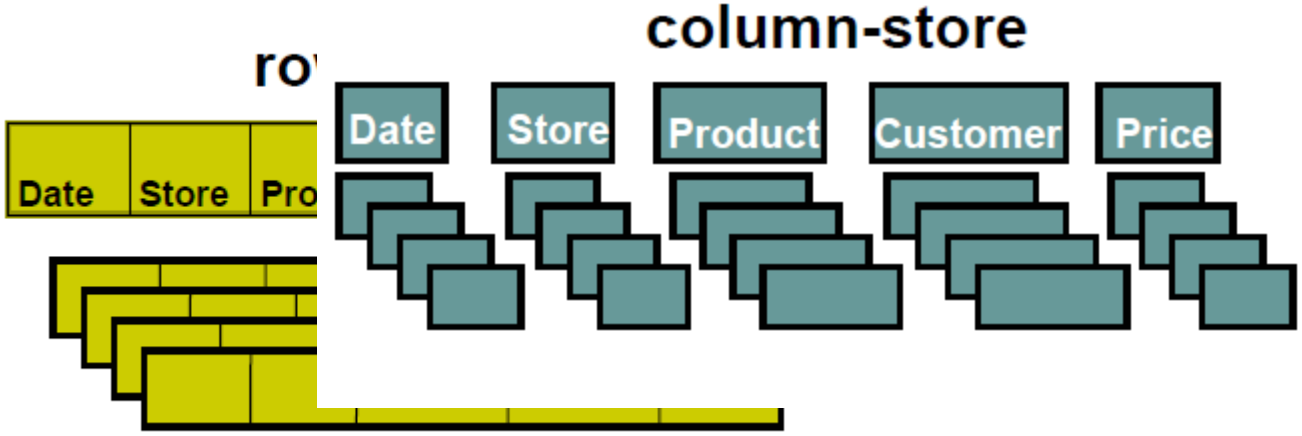
- Database query languages must be translated to programs that can execute
- There may be different ways to do so ➔ optimization
- How can you improve performance?


Data Modeling to Physical Implementation (1)

- At a certain point, the constructs of a data model will have to be mapped/represented in some electronic format (bits/bytes)
- Different storage types with different constraints:
  - Tapes
  - Disks
  - Solid-State Drives
  - Main-memory

Data Modeling to Physical Implementation (2)

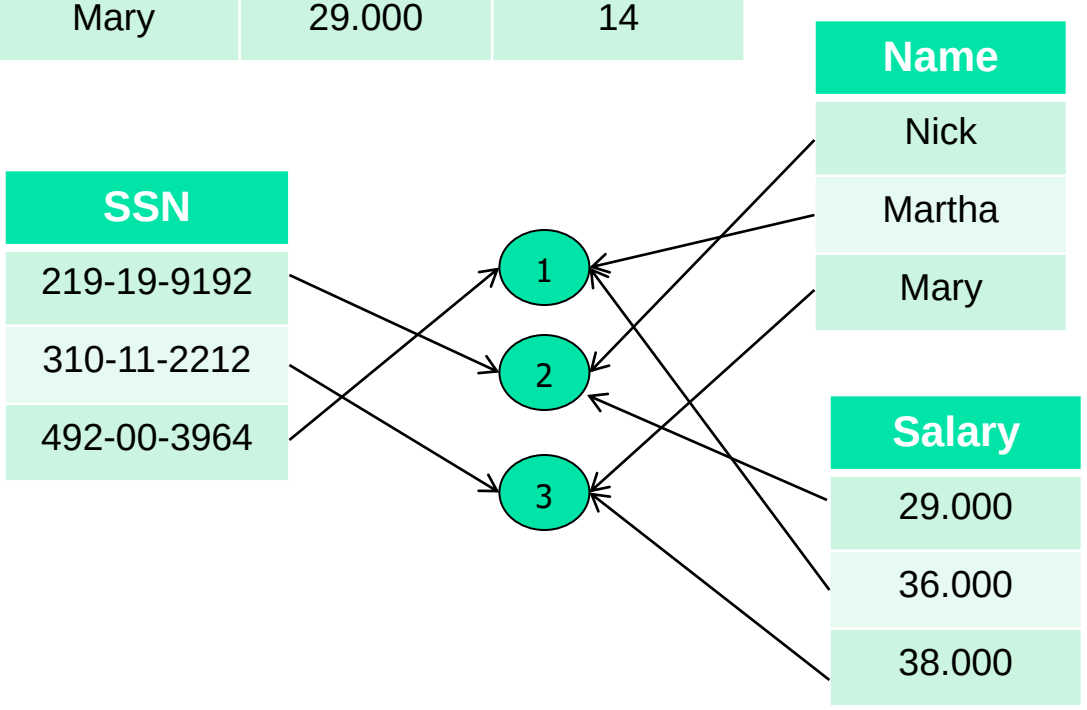
- Different alternatives, depending on the data model, application profile and storage type
- Example: relational model (tables)
- A table can be stored row-wise or column-wise



Data Modeling to Physical Implementation (3)

- Sto

SSN	Name	Salary	DeptCode
492-00-3964	Martha	36.000	11
219-19-9192	Nick	38.000	12
310-11-2212	Mary	29.000	14

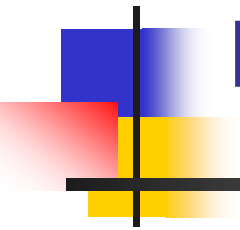



Layer Independence

- Physical Data Independence: the ability to modify the physical schema without changing the logical schema
- Example:
  - Logical level: relations (tables) / SQL
  - Physical level: the table can be represented in disk/main-memory, row-oriented/column-oriented


Issues for Data Management Systems

- Data model
- Query languages
- Evaluation algorithms
- Physical representation
- Performance
- Atomicity, Consistency, Isolation, Durability
- Accessibility
- Security



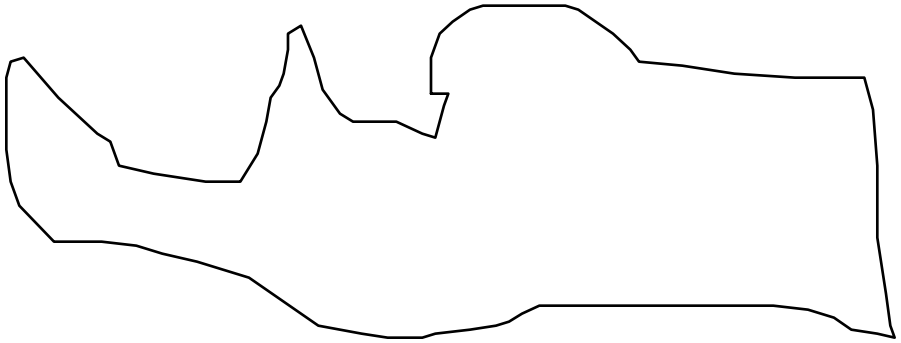
Data Modeling

What is a Model? (1)

- A set of structures/constructs/concepts to describe a real-world situation.

formal model

real-world situation

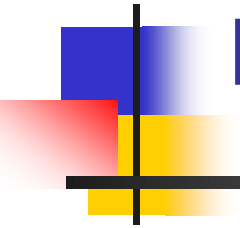



What is a Model? (2)

- Models in all areas:
  - Mathematics, physics, chemistry, biology, linguisitics, poetry
- Why?
  - to store and describe data
  - to explain and predict behavior
  - simulation


What is a Data Model?

- A set of constructs to describe
  - data
  - relationships between data
  - data semantics
  - constraints on data

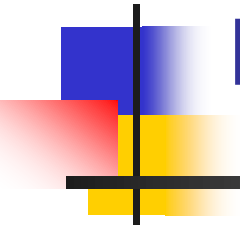


Data Modeling

Entity-Relationship Model

Entity-Relationship Diagrams

- Link (adopted by “Database Systems Concepts” by Silberschatz, Korth and Sudarshan)



Data Modeling

Relational Model

Relational Model

- Link (adopted by “Database Systems Concepts” by Silberschatz, Korth and Sudarshan)
- Example:

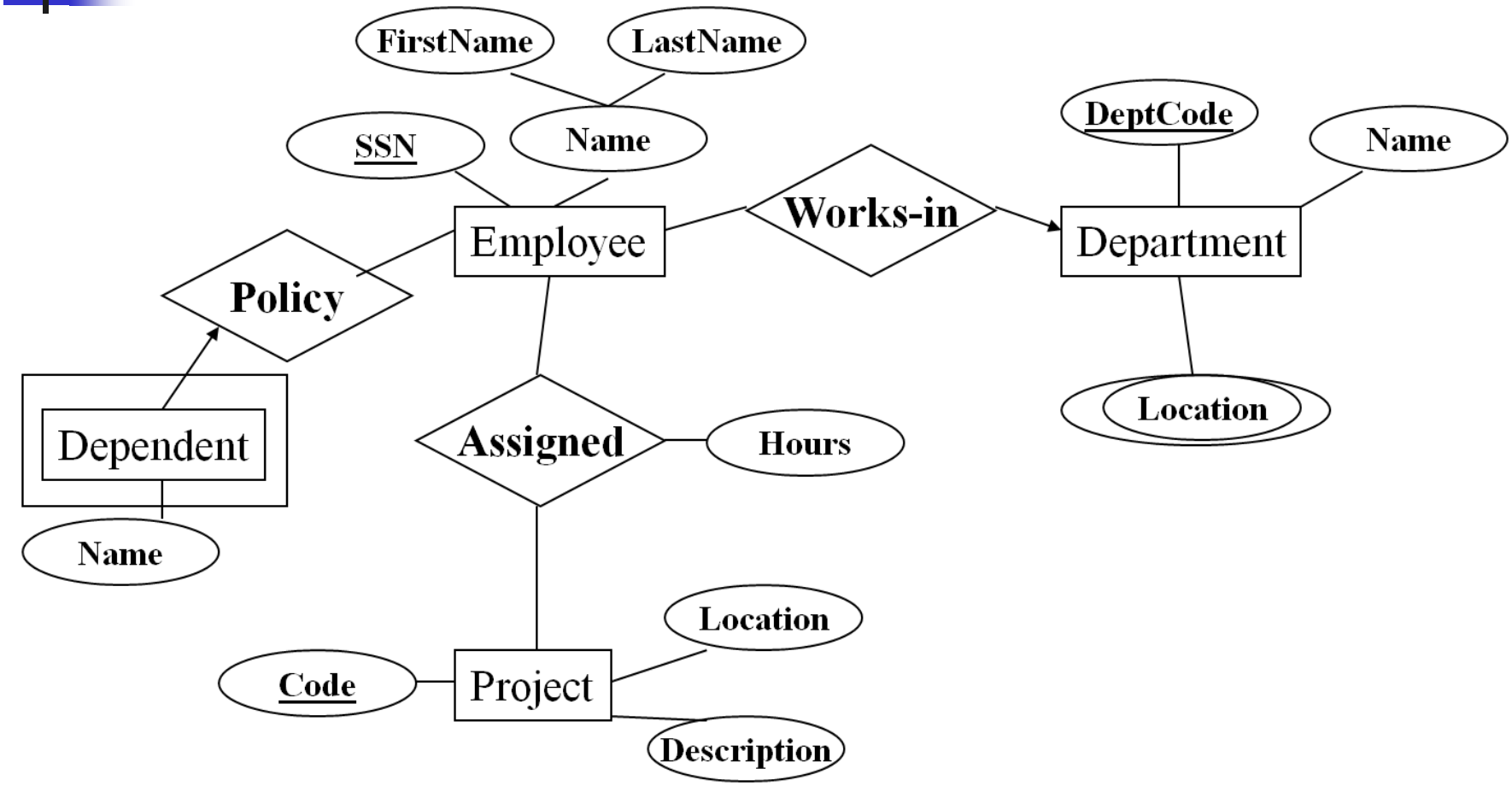
Employee (SSN, FName, LName, Gender, DeptCode)

Dept(Code, Name, Location, Budget)


From ER Design to Relational Schema (1)

- Problem: how to use tables to represent:
  - Strong entities
  - Weak entities
  - Relationships of different cardinalities
  - Different kind of attributes
- Answer: use the rules of the following slides, BUT… it's simple!

From ER Design to Relational Schema (2)




From ER Design to Relational Schema (3)

- Strong entities with single-valued attributes
  - Just a relation, attributes become columns
  - Example: Project (Code, Location, Description)
- Weak entities with single-values attributes
  - Problem: no key to identify rows
  - Think: a weak entity acquires existence from a strong entity
  - A relation having columns the attributes of the weak entity + the primary key of the identifying entity. The primary key of this relation is the primary key of the identifying relations + the discriminator
  - Example: Dependent (SSN, name)


From ER Design to Relational Schema (4)

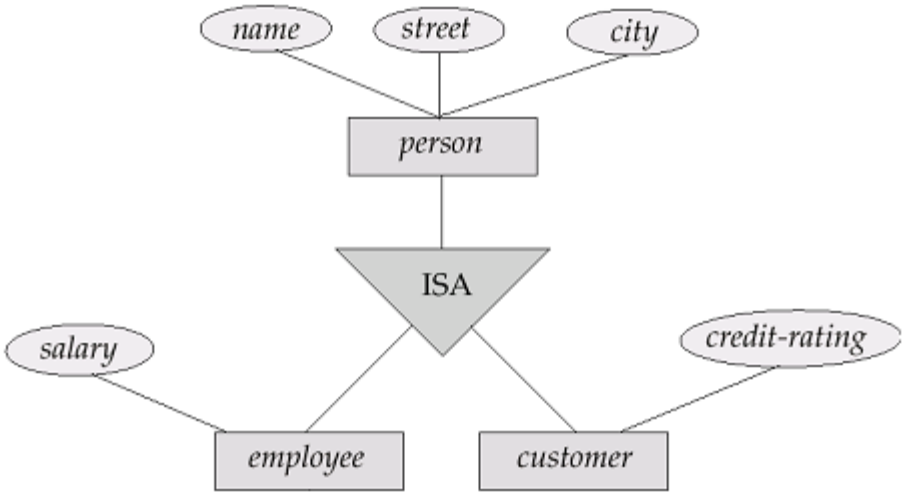
- Attributes:
  - Single-valued attributes become…
    - simple columns
  - Composite attributes become…
    - a set of columns, those at the lowest level
    - example: Employee (SSN, FirstName, LastName)
  - Multi-values attributes become… this is more difficult…
    - a new table, with the primary key and the attribute
    - example: Locations (DeptCode, Location)

From ER Design to Relational Schema (5)

- Relationships:
  - a relationship connects an entity to another entity → in a table that means a row to another row. How can you represent this?
  - A row is identified by the primary key of the table → a pair of primary keys. Create a table with two columns
  - example: Assign (SSN, Code, Hours)
  - however… if the relathionship is 1-N or N-1 no need for a table
  - one can represent the relationship adding a column to the table of the 1-side
  - example: Employee (SSN, FirstName, LastName, DeptCode)

From ER Design to Relational Schema (6)

- Generalization/Specialization: one table per entity?



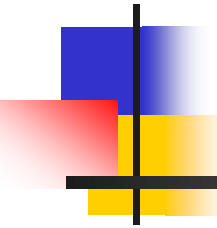
person (<u>name</u>, street, city)

employee (<u>name</u>, salary)

customer (<u>name</u>, credit-rating)


Other Data Models

- Semi-structured: XML, JSON
- Key-value based: dictionaries
- Graph-based models: graphs
- Object-oriented



Query Languages

Using Programming Languages and Flat Files


Programming Languages (1)

- What is a program?
  - a set of instructions executed sequentially that:
    - manipulates data
    - controls the flow
    - performs input/output
  - example (pseudo-code):

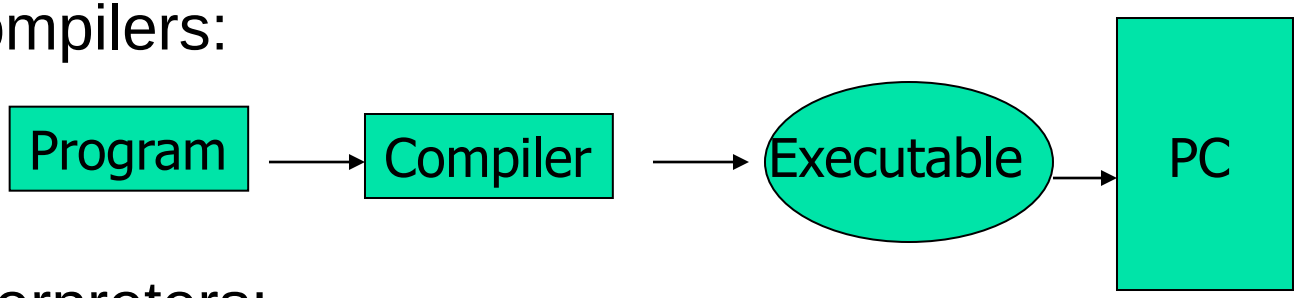
```
main-program {
    integer i;
    input(i);
    if (i<10)
        print(“your number is less than 10”);
    else
        print(“your number is greater than 10”);
}
```

Programming Languages (2)

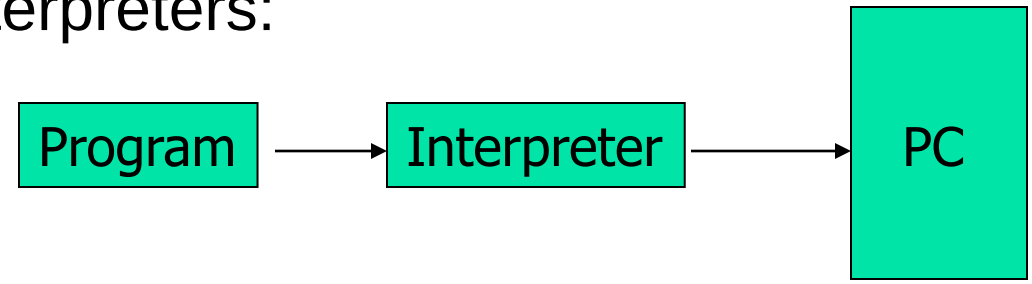
High level languages (need translation, less efficient) - humans

Low level languages (0s and 1s, more efficient) - machine

- Compilers:



- Interpreters:

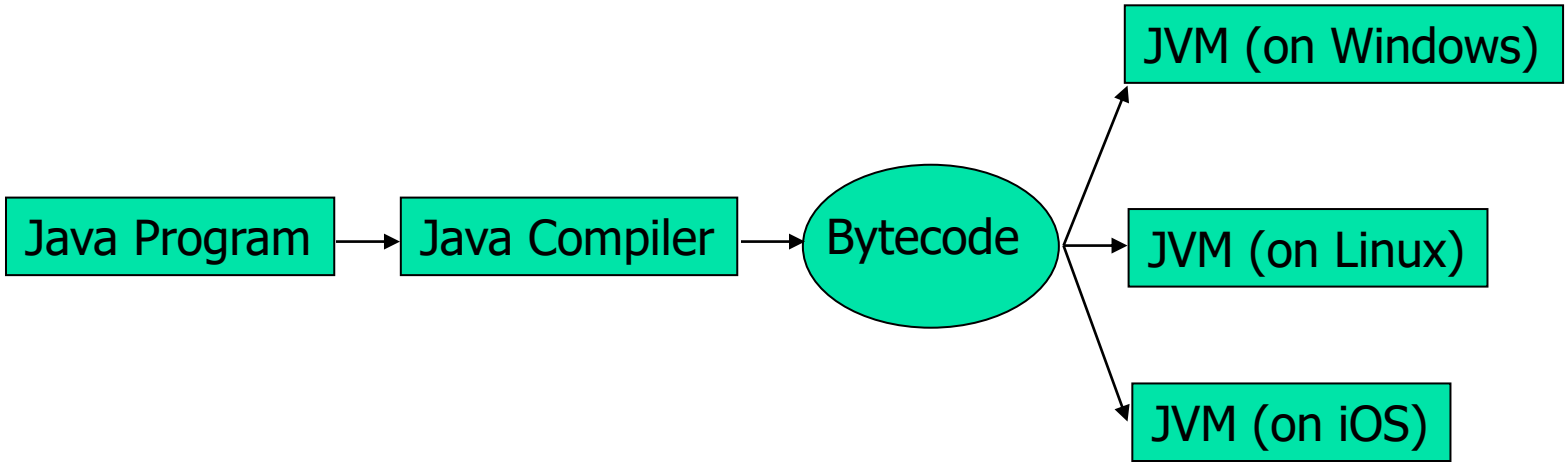


Programming Languages (3)

- Java, a hybrid solution, partially compiled, partially interpreted.
- It defines an “ideal” machine (java virtual machine, JVM) and generates “machine code” (bytecode) for this machine (compiled phase.)
- This machine (JVM) sits on top of another, real, machine and maps bytecode to machine code, specific for that real machine, during execution (interpretation phase.)
- Portable, secure, good performance.

Java – Compilation

- Install the JVM for your machine and you are set to go. Just get the bytecode (translated in any machine, .class file) and execute it at your platform.




Java - Data Types

- byte: integer, 8bit, -128 to 127, stream of data
- short: integer, 16bit, -32768 to 32767
- int: integer, 32bit, -2147483648 to 2147483647
- long: integer, 64bit, large
- float: real, 32bit, single-precision
- double: real, 64bit, double-precision
- boolean: true/false
- char: a (international) character (e.g. ‘θ’) - int


Java - Variables

- Variables must have a valid data type
  - e.g. byte b, c;
  - e.g. short s;
  - e.g. float f;
  - e.g. char ch1, ch2;
    ch1 = 88;
    ch2 = ‘Y’;


Java - Expressions

- Expressions manipulate data items:
  - arithmetic expressions: a * 2, 100 + (b / 6)
  - boolean expressions: (a>10) && (b<c)
- expressions evaluate to a data type

  - a > 10 - boolean (i.e. true/false)

  - int k=10; - int

    s = r * 2 - double
- constituents of an expression must be compatible


Java – Arrays

- An array is a group of *same*-type variables.
- First need to *declare* an array
  - e.g. int month_days[ ];
- Then you need to *create* the array
  - e.g. month_days = new int[12];
- Can access a specific one by an index
  - e.g. month_days[0]=31;
    month_days[1]=28; etc.


Java – Strings

- Strings are sequences of characters
  - e.g. “Hello World”
- What about this data type?
- In C is an array of characters. NOT in Java.
- A string is an ***object*** in Java. But for most practical applications, you can think of it as a variable.


Java - Controlling the Flow

- Control statements
  - selection statements – if, switch
  - iteration statements – while, do-while, for
  - jumping statements – break, continue

Java - Selection Statements - If

- IF general form:

  if (condition) statement1;

  else statement2;

- condition must be a boolean expression - evaluate to true/false, not in some integer
- e.g. if (a<b) a=0;

  else b=0;


Java – Iteration Statements - For

- For statement:
  - for (initialization; while cond; increment) {

    …

    }

```
class exampleFor {
   public static void main(String args[]) {
      int a, b;

      for(a=1, b=4;  a<b;  a++, b--) {
         System.out.println("a = " + a);
         System.out.println("b = " + b);
      }
   }
}
```


Java – Classes and Methods (1)

- Java is an object-oriented programming language
- A program is a class: data + methods
  - main() is a special method that executes first by default
- A program may import other classes and use their data structures and methods: this is how everything is done in Java.


Java – Classes and Methods (2)

- Define a class called Box (to represent a box + some methods to operate on a box)

```
class Box {
  double width;
  double height;
  double depth;

  // compute and return volume
  double volume() {
    return width * height * depth;
  }

  // sets dimensions of box
  void setDim (double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
}
```


Java – Classes and Methods (3)

- Now another program can use the Box class

```
class ExampleBox {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // initialize each box
    mybox1.setDim(10, 20, 15);
    mybox2.setDim(3, 6, 9);

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

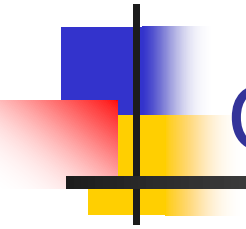

Flat Files

- What is a flat file?
  - text format
  - use of delimiters, possibly multiple within the same line
  - consisting of lines
- Example:
  - FlatFileExample.txt
  - first name | last name | age | salary | department code

```
Nick,Atkinson,32,8700,4
```


Programming Langs & Flat Files – Example

- Write a java program to read FlatFileExample.txt and print out a line per employee and totals at the end.
- employees.java
- Notes:
  - iterate over all lines in the file – cannot select what you want
  - *processing is always line by line*
  - format the output exactly as you want, or use it any way you like
  - peculiar computations (e.g. some weird statistical metric)
  - procedural flexibility

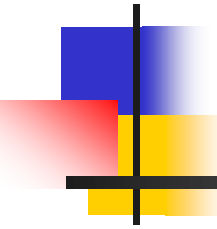


Query Languages

A Theoretical
Approach: Relational Algebra


Relational Algebra

- Slides – adopted by “Database System Concepts”, Silberschatz, Korth, Sudarshan, 6th edition.



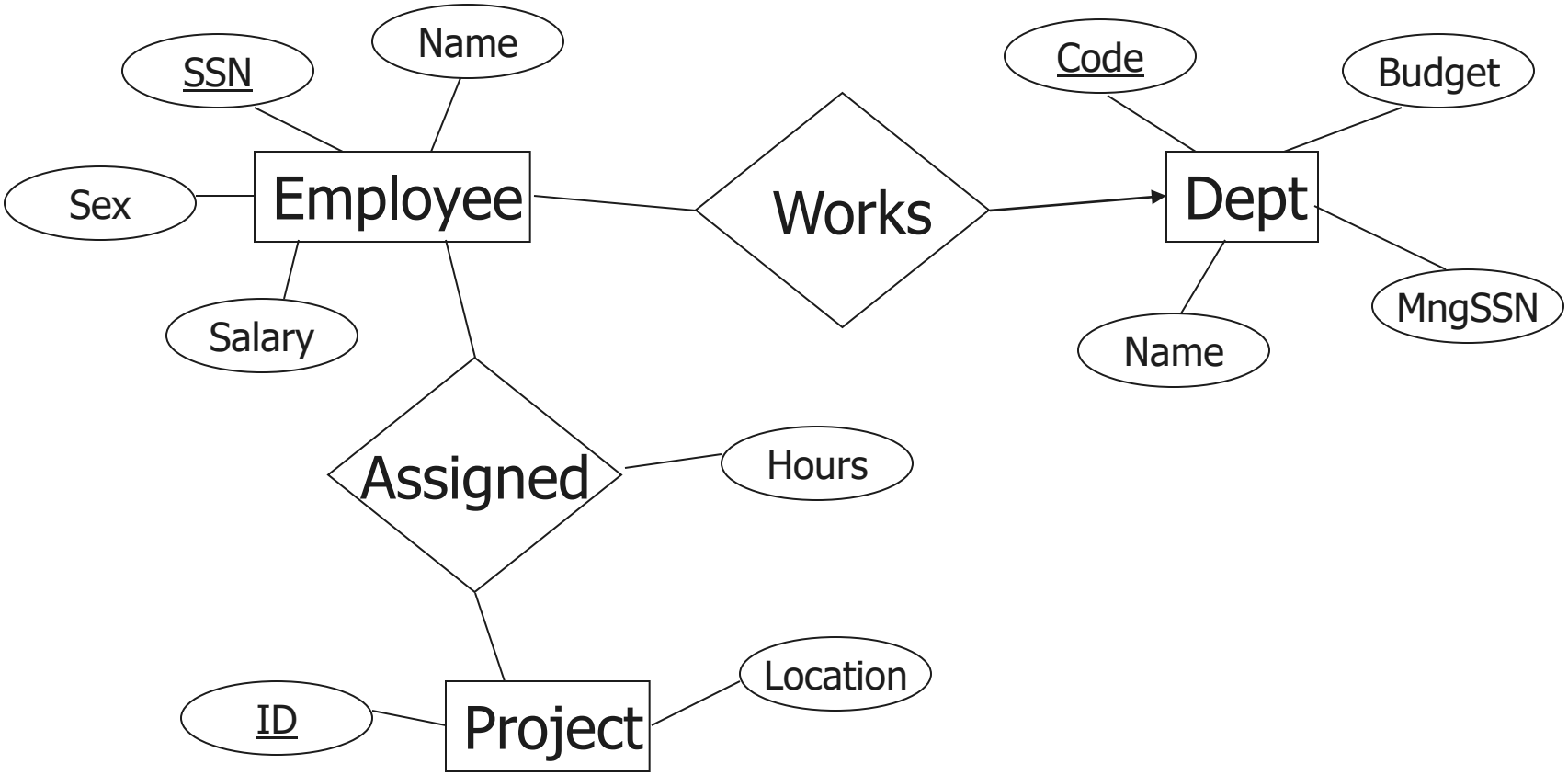
Query Languages

SQL: Basic Statements, Nested Queries, Aggregation, Procedural SQL, Triggers, Cursors, Stored Procedures


SQL - General

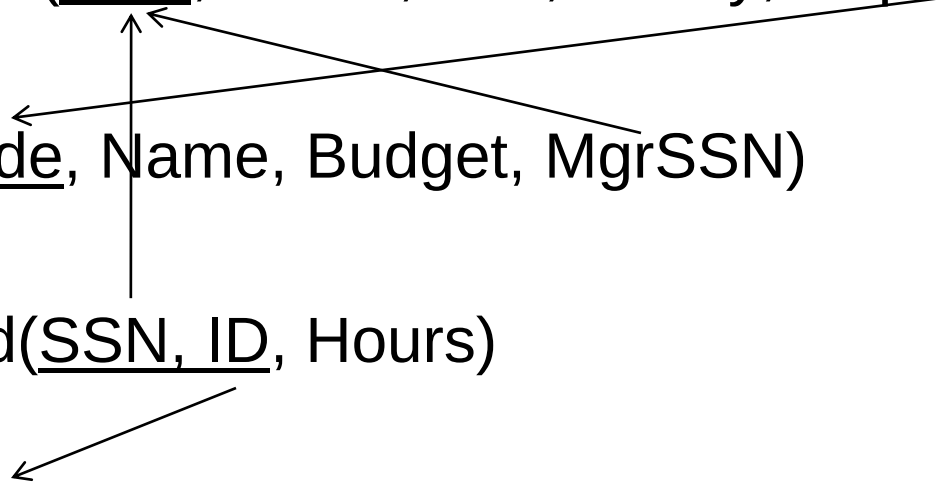
- SQL: Structured Query Language.
- Declarative - specify **what** you want, not **how**.
- Translates to relational algebra; the expression is then optimized and evaluated.
- Developed in IBM, late 70s.
- Running Example: employees work in departments and are involved in projects.

Running Example: E-R diagram



Running Example: Tables

- Employee(SSN, Name, Sex, Salary, DeptCode)
- Dept(Code, Name, Budget, MgrSSN)
- Assigned(SSN, ID, Hours)
- Project(ID, Location)




Basic SQL statement

```
SELECT A1,A2, …,An
FROM T1, T2, …, Tk
WHERE condition.
```

- Select columns `A1,A2, …,An`
  from tables `T1, T2, …, Tk`
  of those records that satisfy the `condition`.
- Relational algebra:

$$\pi_{A_1, A_2, .., A_n} (\sigma_{\text{condition}} (T_1 \times T_2 \times \ldots T_k))$$

SQL – Example 1 (select)

- Give me all attributes of the employees (no `where`):

```
select SSN, Name, Sex, Salary, DeptCode
from Employee
```

or

```
select *
from Employee
```


SQL – Example 2 (functions, distinct)

- Show the name (in capital letters) and salary (in euros):

```
select UCASE(Name), Salary * 1.12
from Employee
```

There are functions for string/number/date manipulation. Could appear in `SELECT` or `WHERE` clauses

- Give me the distinct names of all the employees:

```
select distinct Name
from Employee
```


SQL – Example 3 (where)

- Show the social security number of all female employees with salary more than 50,000:

```
select SSN
from Employee
where salary > 50000 AND sex='f'
```

- where month(dateOfBirth)=3
- date()

SQL – Example 4 (join)

- Show the SSN for all employees along with the name of their department:

```
select Employee.SSN, Dept.Name
from Employee, Dept
where Employee.DeptCode=Dept.Code
```

```
select E.SSN, D.Name
from Employee as E, Dept as D
where E.DeptCode=D.Code
```

SQL – Example 5 (join)

- Show the name and salary of all employees who work in some project located in ‘Boston’:

```
select    Name, Salary
from      Employee as E, Assigned as A,
          Project as P
where     E.SSN=A.SSN AND A.ID=P.ID AND
          Location LIKE '%Boston%'
```

LIKE operator is used for string matching (wildcards)

SQL – Example 6 (self-join)

- For each department D, show its code and all the codes of the departments having budget lower than D's budget:

```
select D.Code, G.Code
from Dept as D, Dept as G
where D.Budget > G.Budget
```

SQL – Example 7 (order by)

- Show the SSN and the salary of all employees, in ascending order of their salary:

```
select SSN, salary
from Employee
order by salary asc
```

asc/desc for ascending/descending order

SQL – Example 8 (aggregation)

- Find the minimum, maximum and average salary of department 121:

```
select min(salary),max(salary),avg(salary)
from Employee
where DeptCode=121
```

Aggregate functions: min, max, avg, sum, count
Regular attributes cannot appear in the `select`

SQL – Example 9 (group by)

- Show the average salary of each department:

```
select DeptCode, avg(salary)
from Employee
group by DeptCode
```

- Show the average salary by department and sex:

```
select DeptCode, Sex, avg(salary)
from Employee
group by DeptCode, Sex
```

SQL – Example 10 (join, group by)

- Show the total number of hours assigned to projects per department code:

```
select DeptCode, sum(Hours)
from Employee as E, Assigned as A
where E.SSN=A.SSN
group by E.DeptCode
```

SQL – Example 11 (having)

- Show the average salary of each department, but only if it has more than 3 employees:

```
select DeptCode, avg(salary)
from Employee
group by DeptCode
having count(*) > 3
```

SQL – Example 12 (minus)

- Show the SSN of those employees that are *not* managers:

```
(select SSN
 from Employee)
minus
(select MgrSSN
 from Dept)
```

SQL – Example 13 (union)

- Show the SSN of employees who make more than 80000 and the SSN of employees who work more than 30 hours:

```
(select SSN
 from Employee
 where salary > 80000)
union
(select SSN
 from Assigned
 group by SSN
 having sum(Hours) > 30 )
```


SQL – Subqueries (1)

- SELECT… FROM… WHERE
- Idea is to pick those rows that satisfy a condition
  - If θ(r) is true, then select the row
- So far this condition involves attributes of the row
  - Salary > 60000 AND Sex = ‘f’
- However, it could be more complex and involve sets
  - Salary > 60000 AND DeptCode in {132, 451}
  - Salary greater than the salaries of all managers ?
  - Salary > all (SELECT … FROM … WHERE)

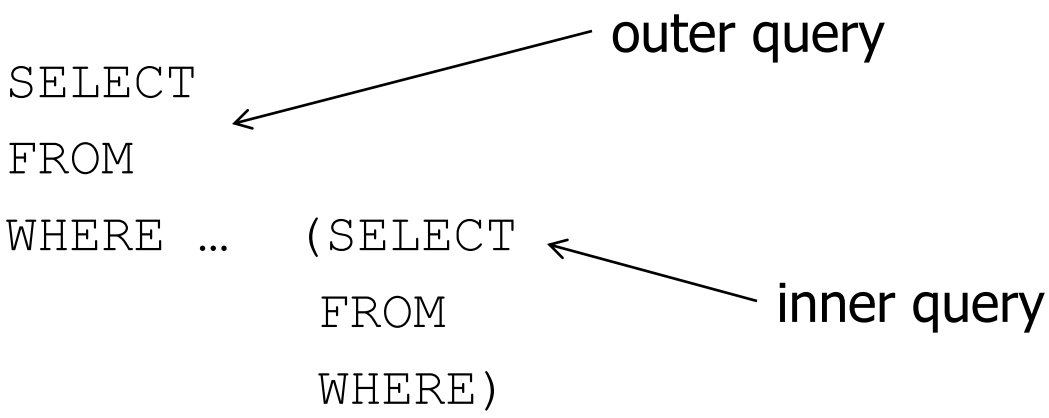
SQL – Subqueries (2)

- Find the SSN and Name of all employees that work in a department with budget > 500000

```
select SSN, Name
from Employee
where DeptCode in  (select Code
                    from Dept
                    where Budget > 500000 )
```

SQL – Subqueries (3)

- General format:




SQL – Subqueries (4)

- WHERE <value> in / not in A
- WHERE <value> < <= some A
  > >=
  = <>
- WHERE <value> < <= all A
  > >=
  = <>
- WHERE exists/ not exists A
- WHERE unique/ not unique A

A is a select-from-where statement.

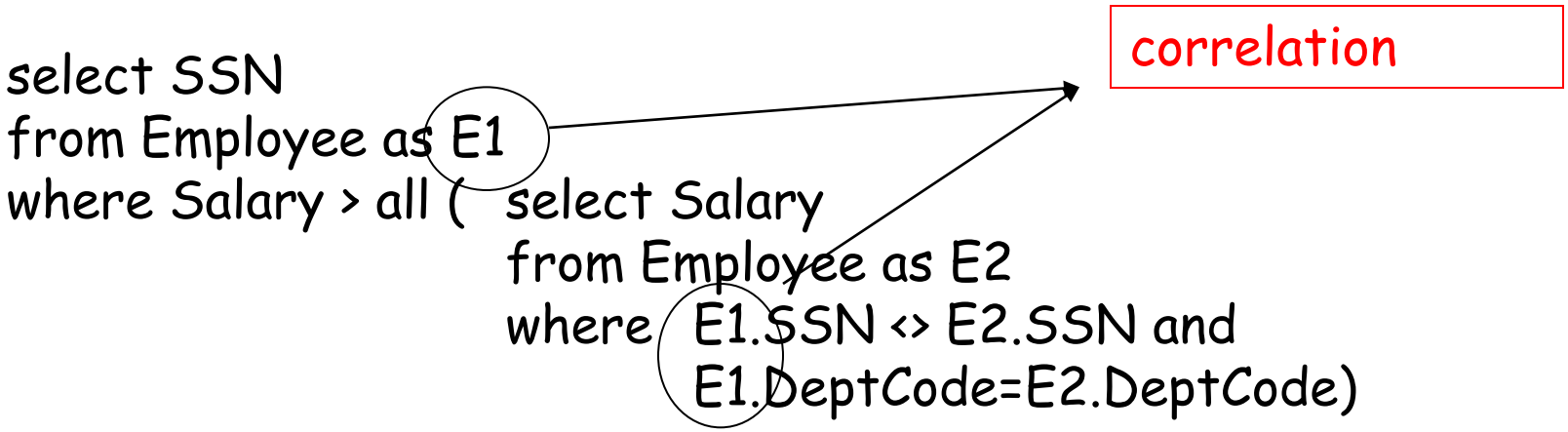
SQL – Subqueries (5)

- Show the SSN of employees who make more than all managers:

```
select SSN
from Employee
where salary > all (select salary
                    from Employee
                    where SSN in (select MgrSSN
                                  from Dept))
```

SQL – Subqueries (6)

- For each department, find the SSN of the employee with the highest salary.



SQL – Schema Definition (1)

- One can use SQL statements to create/alter/delete the schema of a database
- Create a table:

```
create table Employee (
  SSN char(11) not null,
  Name varchar(40),
  Age int,
  Salary money,
  DeptCode int,
  primary key (SSN),
  foreign key (DeptCode) references Dept)
```

- Delete a table (from the schema):

```
drop table Employee
```

SQL – Schema Definition (2)

- Views are virtual tables, part of the schema, representing the result of an SQL statement (schema)

```
create view EmpDept (SSN, EName, DName, Salary, MgrSSN) as
select E.SSN, E.Name, D.Name, E.Salary, D.MgrSSN)
from Employee as E, Dept as D
where E.DeptCode=D.Code
```

- The answer of the SQL query may or may not be kept
- If the answer is kept ➔ Materialized views
- If we have materialized views, we should have view maintenance.


SQL – Example 13

- For each employee, express his/her salary as percentage of his/her department's average salary.

```
create view V(DeptCode,avgSalary) as
select DeptCode, avg(Salary)
from Employee
group by DeptCode

select SSN, (Salary/avgSalary)
from Employee E, V
where E.DeptCode=V.DeptCode
```

SQL – Example 14

- Show the SSN of the employees who work longer hours than the average

```
select SSN, avg(Salary)
from Assigned, (select avg(Hours) as avgHours
                from Assigned) as Temp
where Hours > Temp.avgHours
```

SQL – Database Updates (1)

- One can use SQL to insert/delete/update data in a DB
- Insert data:

```
insert into Employee
values('201-20-1199','John James',32,67000,4)

insert into Employee(Name, SSN, Salary, Age, DeptCode)
values('John James','201-20-1199',67000,32,4)

insert into Employee
select… from… where (an SQL statement)
```


SQL – Database Updates (2)

- Delete data:

```
delete from Employee
where DeptCode=4

delete from Employee
where Salary > all (select avg(Salary)
                    from Employee)
```

- Exercise caution when you delete from a table that another one references (why?)
  - alternatively, declare in the schema “on delete cascade”


SQL – Database Updates (3)

- Update data:

```
update Employee
set Salary = Salary * 1.05
where DeptCode=4
```

```
update Employee
set Salary = Salary * 1.05
where Salary < all (select avg(Salary)*0.9
                    from Employee)
```

- Exercise caution when you update a table that another one references (why?)
  - alternatively, declare in the schema “on update cascade”


Procedural SQL (1)

- In many cases, for several reasons (performance, special handling, recursion), it is necessary to extend SQL with procedural features (loops, variables, ifs, etc.)
- Examples:
  - Oracle: PL/SQL
  - SQL Server: T-SQL
- Programs that *contain* SQL statements

Procedural SQL (2)

- An Example in PL/SQL

```
DECLARE
  v_LoopCounter BINARY_INTEGER := 1;
BEGIN
  LOOP
    INSERT INTO temp_table (num_col)
      VALUES (v_LoopCounter);
    v_LoopCounter := v_LoopCounter + 1;
    EXIT WHEN v_LoopCounter > 50;
  END LOOP;
END;
```

Cursors (1)

- A cursor corresponds to the answer of an SQL statement (i.e. is a table) that can be processed sequentially, row by row, in a top-down fashion.

```
declare EmpSalary
for select SSN, Name, Salary
from Employee
where age>30
order by Salary desc
```


SSN	Name	Salary
121-11-3525	Ann	152000
219-02-2212	John	148000
178-00-1281	Nick	145000
377-01-7833	Mary	132000
…	…	…

Cursors (2)

- Cursors are declared within a procedural SQL program
- Declare a cursor, open a cursor, close a cursor
- Fetch a row: reads in current record and advances cursor to the next row.

```
fetch EmpSalary into @ssn, @name, @salary
```


Stored Procedures

- A stored procedure is a named, parameterized SQL statement that is stored and can be executed by using its name and actual values. Faster than writing plain SQL because optimization/compilation has already been done.

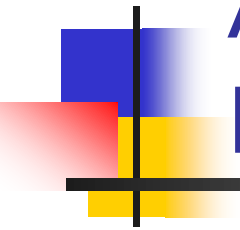
```
CREATE PROCEDURE deptEmployees
    @code int
AS
    SELECT SSN, Name, Age, Salary
    FROM Employee
    WHERE  DeptCode = @code

To execute the stored procedure:

EXECUTE deptEmployees 4
```

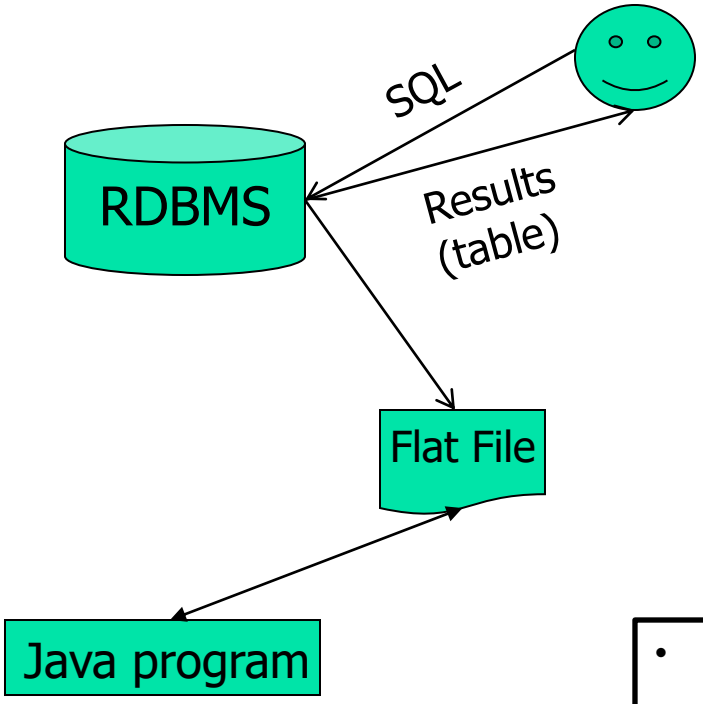

Triggers

- In many cases we want to do the following:
  - In case of an *event*, check for a *condition* and if true, *act*.
- These are called triggers or ECA rules. Triggers make a database an *active* database.
- Example: In case of a salary increase (event), check if the total salaries of that department's employees is greater than the department's budget and if yes, do nothing. Otherwise, proceed with the insert.



APIs for Accessing Data in Relational Systems

Motivation – PL and RDBMS (1)



- What can we do with the results? We rarely want to just see them in the console.
- In most cases we are not even the consumers of the results!
- In most cases programs (written in some programming language) are the consumers of the results (web, excel, etc.)
- How can programs use the results?
- Same question with updating data. We rarely directly inserting data into a RDBMS. Programs do.

- We must find a way so programs can directly access RDBMS using SQL


Motivation – PL and RDBMS (2)

- How programs can directly send SQL statements (to update/retrieve data) to RDBMS?
- Have (once again) an API (a set of methods) so a programming language can access a RDBMS
  - connect to the database providing the username/password
  - send an SQL statement to the RDBMS
  - get the results as a table
    - iterate over the table row by row
    - access row using column names

A First Example: Using Java to Connect

```
Statement s = dbcon.createStatement();
ResultSet rs = s.executeQuery("SELECT * FROM Employee WHERE DeptCode=2");
while (rs.next()) {
    System.out.print(rs.getString("SSN"));
    System.out.print(rs.getString("Name"));
    System.out.println(rs.getDouble("Salary"));
}
```

create an SQL statement

method to execute an SQL statement

the result (a table) is stored in this variable of type ResultSet

column names

fetch next row

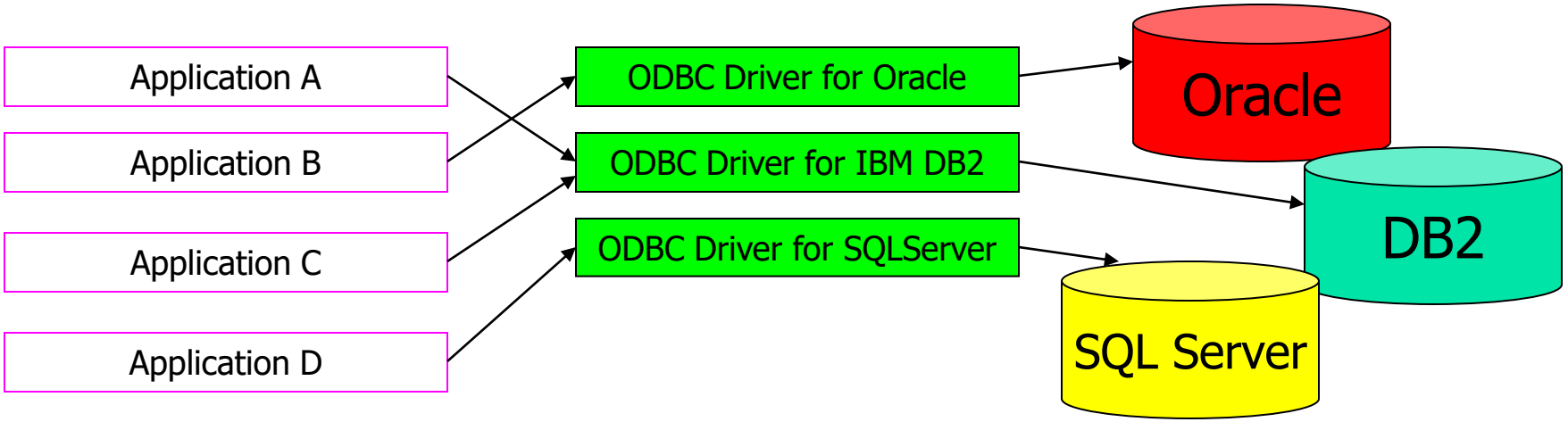

Open Database Connectivity (ODBC) (1)

- Many programming languages
  - C, C++, Fortran, Python, Java, PHP, …
- Many database systems
  - SQL Server, MySQL, Oracle, DB2, Postgres, …
- Do we have to have an API for each programming language, for each database system?
- Can we do something better? Invent a middle layer between programming languages (or, in general, applications) and RDBMS

Open Database Connectivity (ODBC) (2)

- Developed by Microsoft.
- ODBC is a widely accepted generic application programming interface for database access.
- The goal is to make it possible to access *any* data from *any* application, regardless of which DBMS is handling the data.
- ODBC “inserts” a middle layer - called database driver - between the application and the DBMS.

Open Database Connectivity (ODBC) (3)



- An interface between a database and an application, so the application can retrieve data
- Communication between data source and application through a well-defined set of methods


Open Database Connectivity (ODBC) (4)

- In Windows:
  - To open Data Sources (ODBC), click **Start**, point to **Settings**, and then click **Control Panel**. Double-click **Administrative Tools**, and then double-click **Data Sources (ODBC)**.
  - For information about using Data Sources (ODBC), click **Help** in the ODBC **Data Source Administrator** dialog box.

Open Database Connectivity (ODBC) (5)

- ODBC greatly contributed to the growth of relational systems. Prior to ODBC, applications had to rely on several, language-specific, APIs.
- Standardization of *data connectivity* helped innovation and productivity, allowing developers to focus on the core of their ideas.
- Performance is poor - due to the intermediate layer
  - Instead of ODBC, use a DBMS-specific and programming language-specific API to fetch rows in raw format (e.g. Oracle Call Interface - OCI)


A Complete Java Example

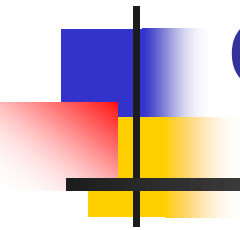
- Database in Access: test.mdb
- Java program: testDB.java


Download MySQL

- Download MySQL Server (Windows/Linux/Mac)
- Windows:
  - https://dev.mysql.com/downloads/windows/
  - Use MySQL Installer to install MySQL Server and Workbench
- Use MySQL Workbench to create a database, write SQL queries, insert/delete/update data, etc.

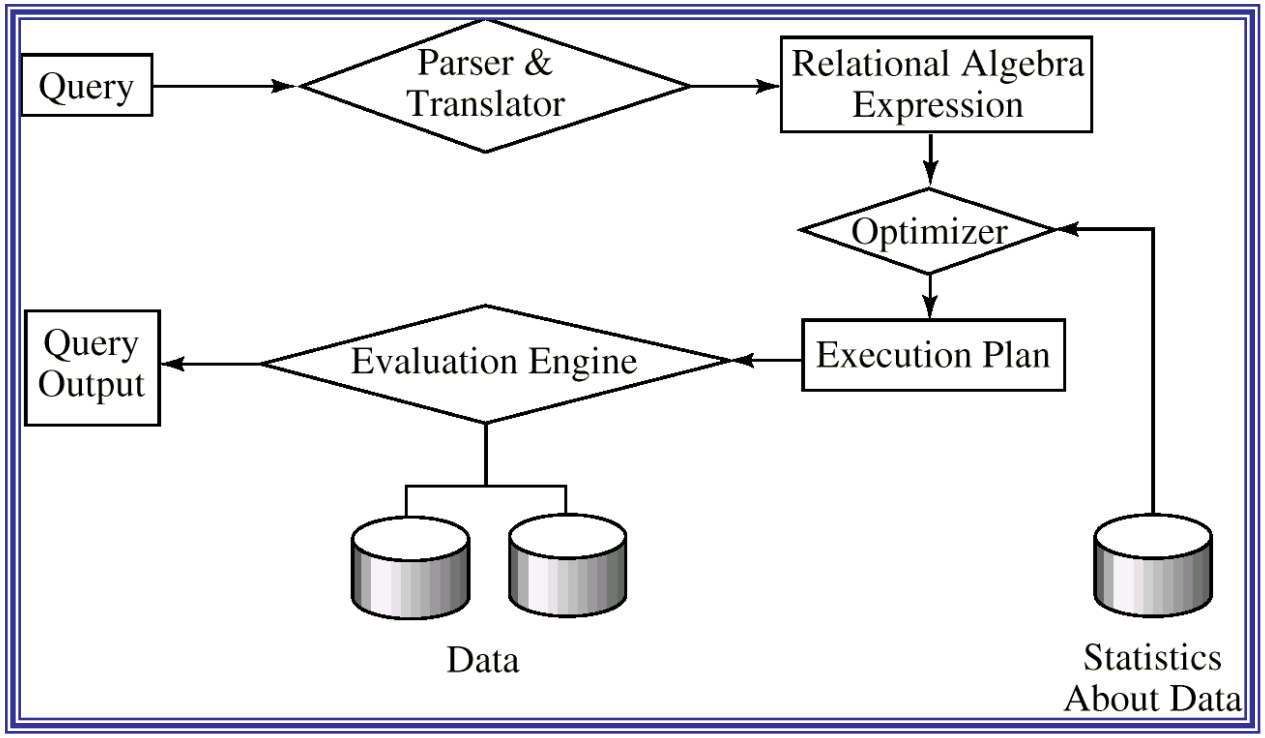

Accessing Data in non-Relational Systems

- Programming paradigms (e.g. MapReduce) [HDFS]
- PigLatin, HiveQL [HDFS]
- XPath [semi-structured, XML]
- MongoDB selectors [semi-structured, JSON]
- Cypher [graph databases]
- Redis QL [Key-Value stores / data structure servers]
- APIs



Query Processing

Query Processing – Big Picture




Query Optimization, Parsing (1)

- Translate the query into its internal form.
- This is then translated into relational algebra.
- Parser checks syntax, verifies relations

Query Optimization, Parsing (2)

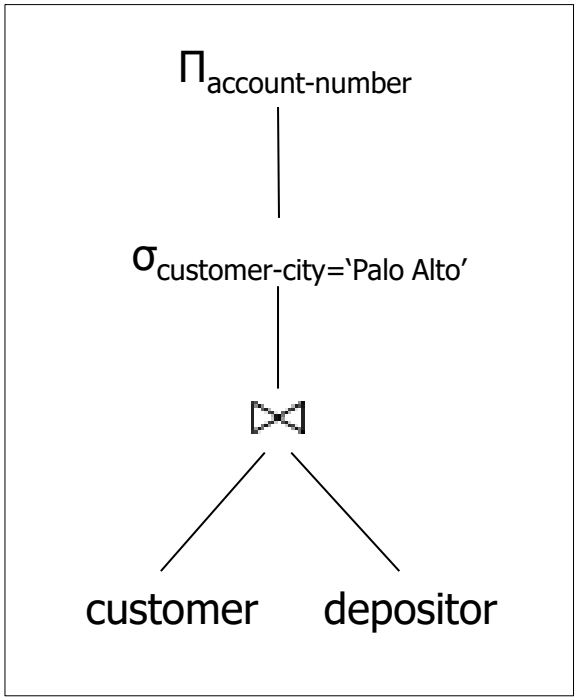
- Show the account number(s) of customers living in Palo Alto.

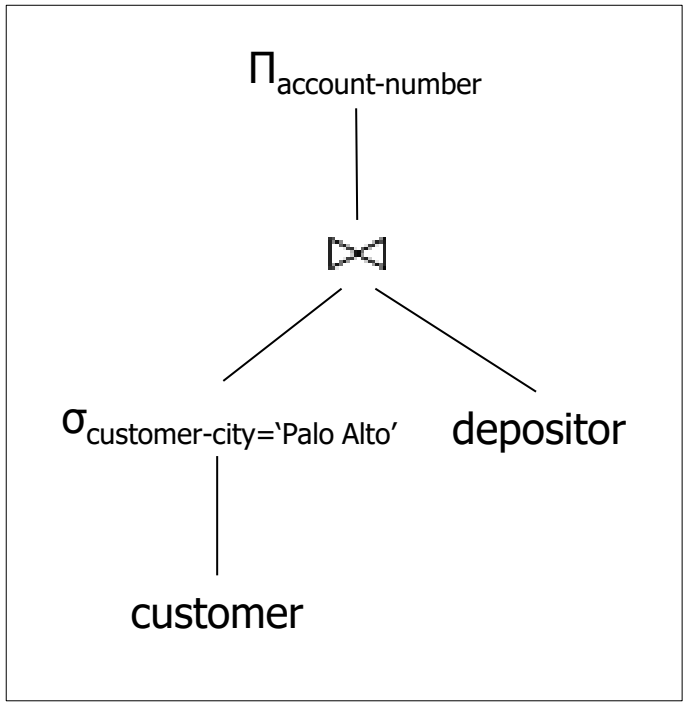
```
SELECT d.account-number
FROM customer as c, depositor as d
WHERE c.customer-id = d.customer-id AND
        c.customer-city='Palo Alto'
```

$$\Pi_{\text{account-number}} (\sigma_{\text{customer-city='Palo Alto'}} (\text{customer} \bowtie \text{depositor}))$$

Query Optimization, Algebra (1)

- Use **equivalence rules** to transform an expression into an equivalent one.





Query Optimization, Algebra (2)

$$\sigma_{\Theta_1 \wedge \Theta_2}(E) = \sigma_{\Theta_1}(\sigma_{\Theta 2}(E))$$

$$\sigma_{\Theta_1}(\sigma_{\Theta_2}(E)) = \sigma_{\Theta_2}(\sigma_{\Theta_1}(E))$$

$$\Pi_{t_1}(\Pi_{t_2}(\ldots(\Pi_{tn}(E))\ldots)) = \Pi_{t_1}(E)$$

$$\sigma_{\Theta}(E_1 \times E_2) = E_1 \triangleright\triangleleft_{\Theta} E_2$$

$$E_1 \triangleright\triangleleft_{\Theta} E_2 = E_2 \triangleright\triangleleft_{\Theta} E_1$$

$$E_1 \triangleright\triangleleft (E_2 \triangleright\triangleleft E_3) = (E_1 \triangleright\triangleleft E_2) \triangleright\triangleleft E_3$$

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \triangleright\triangleleft_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \triangleright\triangleleft_{\theta} (\sigma_{\theta_2}(E_2))$$

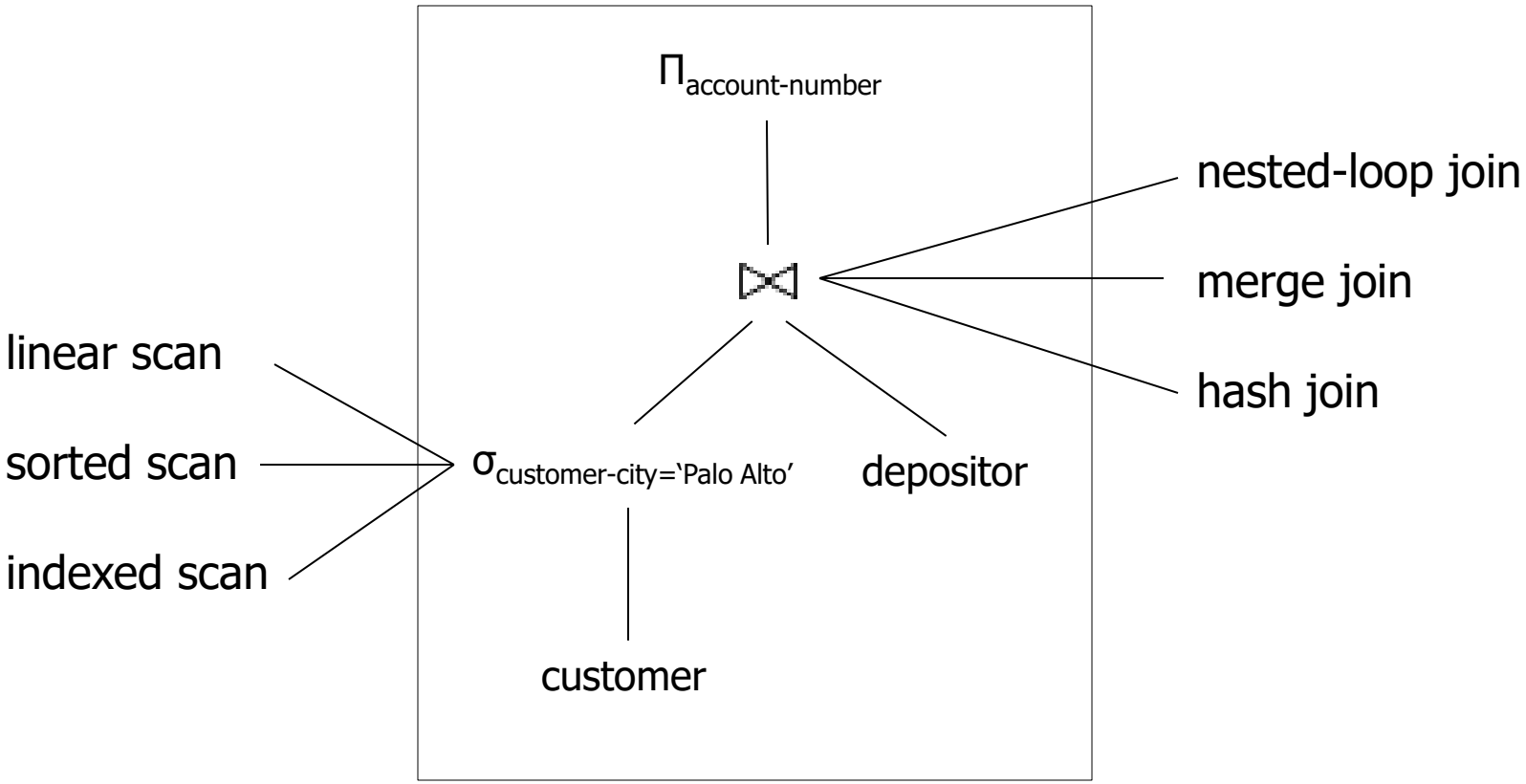

Query Optimization, Implementing Ops (1)

- There are several possible implementation algorithms for each of the operators π, σ, ⋈, U, …
- Each implementation algorithm has an estimated cost
- Measures of query cost to take into consideration:
  - size of tables
  - how many blocks (disk) the table uses
  - file organization
  - indexing (if any), type of index
  - statistics on selectivity, cardinality


Query Optimization, Implementing Ops (2)

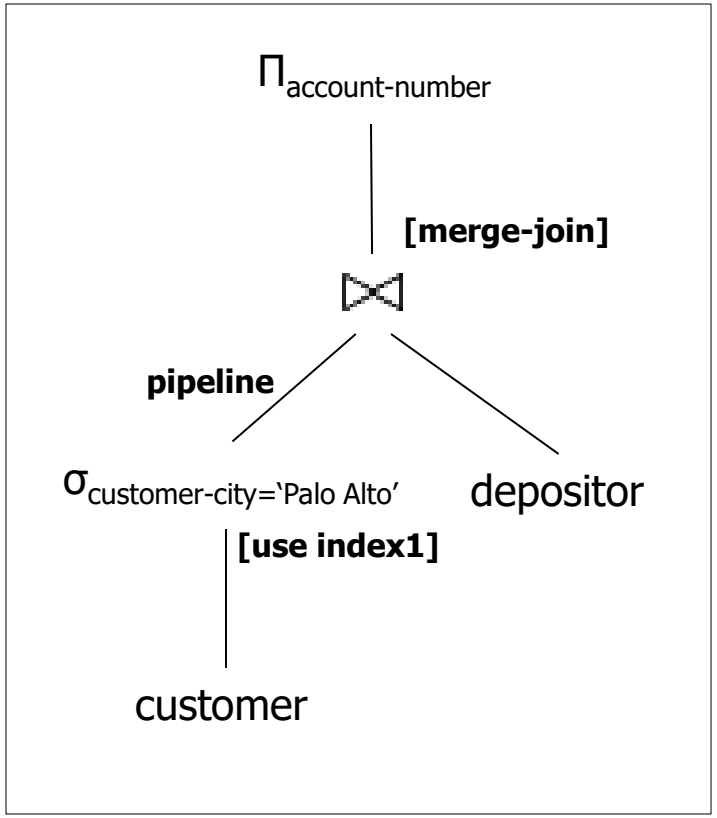
- Selection operator:
  - linear search, binary search, index scan, complex selections
- Join operator:
  - nested-loop join
  - block nested-loop join
  - indexed nested-loop join
  - merge join
  - hash join

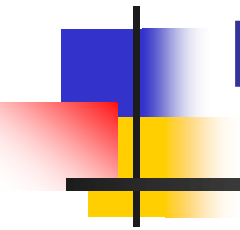
Query Optimization, Implementing Ops (3)



Query Execution, Evaluation Plan

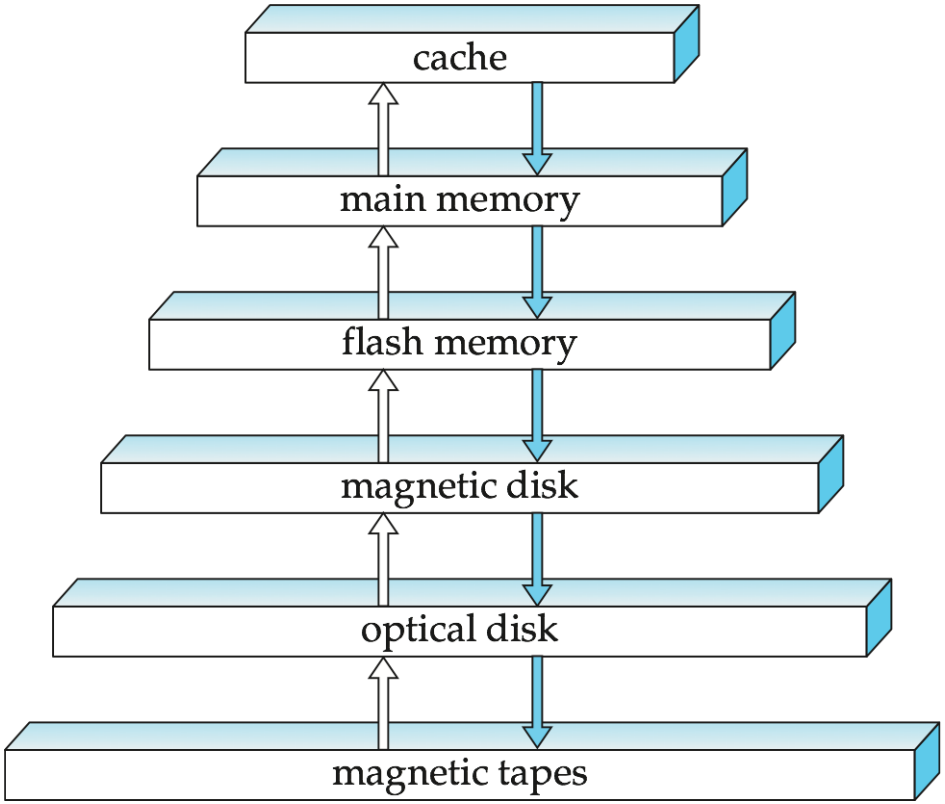
- Try algebraic transformations
- Try alternative implementation algorithms
- Assign cost to each, choose the cheapest one
- Annotated evaluation plan





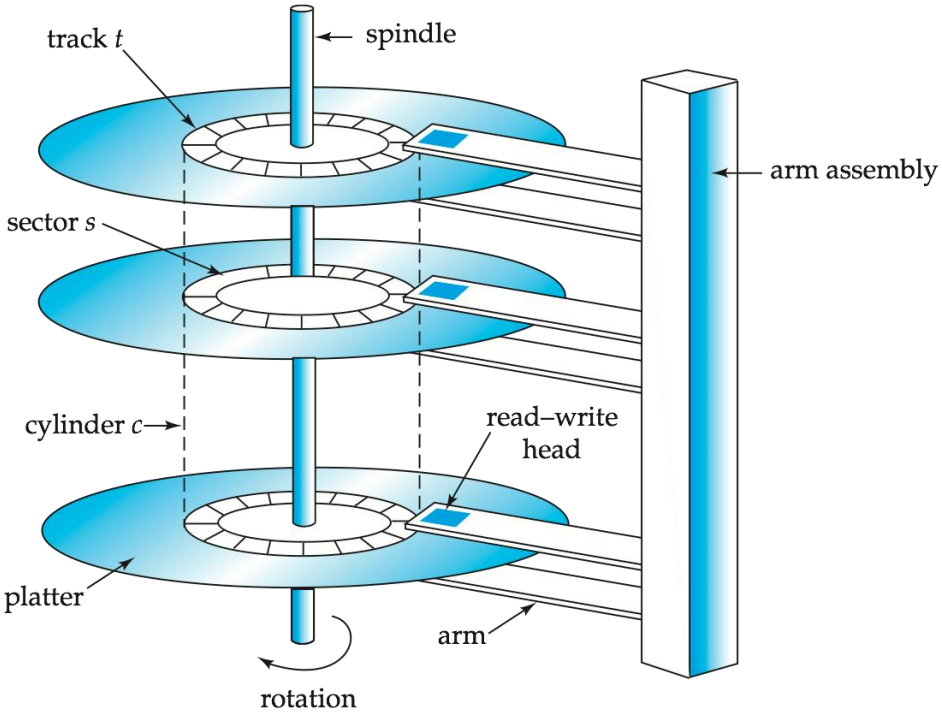
Physical Implementation

Memory-Hierarchy



HDDs (1)

- Platter, Head, Actuator, Cylinder, Track, Sector (physical), Block (logical), Gap
- Time = Seek Time + Rotational Delay + Transfer Time



HDDs (2)

- Typical seek time:
  - 4ms for high end drives
  - 15ms for mobile devices
- Average rotational delay:

HDD Spindle [rpm]	Average rotational latency [ms]
4,200	7.14
5,400	5.56
7,200	4.17
10,000	3.00
15,000	2.00

- Transfer time:
  - "disk-to-buffer" data transfer rate up to 1,030 Mbits/sec


HDDs (3)

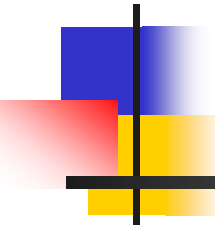
- HDDs are organized (logically) to blocks or pages
  - View: a contiguous set of blocks
  - Size of block varies – usually 512 bytes
  - ‘next block’ concept:
    - blocks on same track, followed by
    - blocks on same cylinder, followed by
    - blocks on adjacent cylinder
  - Files: a set of blocks
    - blocks of a file should be placed sequentially.
    - prefetching can be a big win


SSDs

- Storage is block oriented (not random access)
- https://en.wikipedia.org/wiki/Solid-state_drive


Main-memory

- Random access
- Pointer or references, pointer arithmetic
- Dramatic differences with disk-based / SSD-based implementations

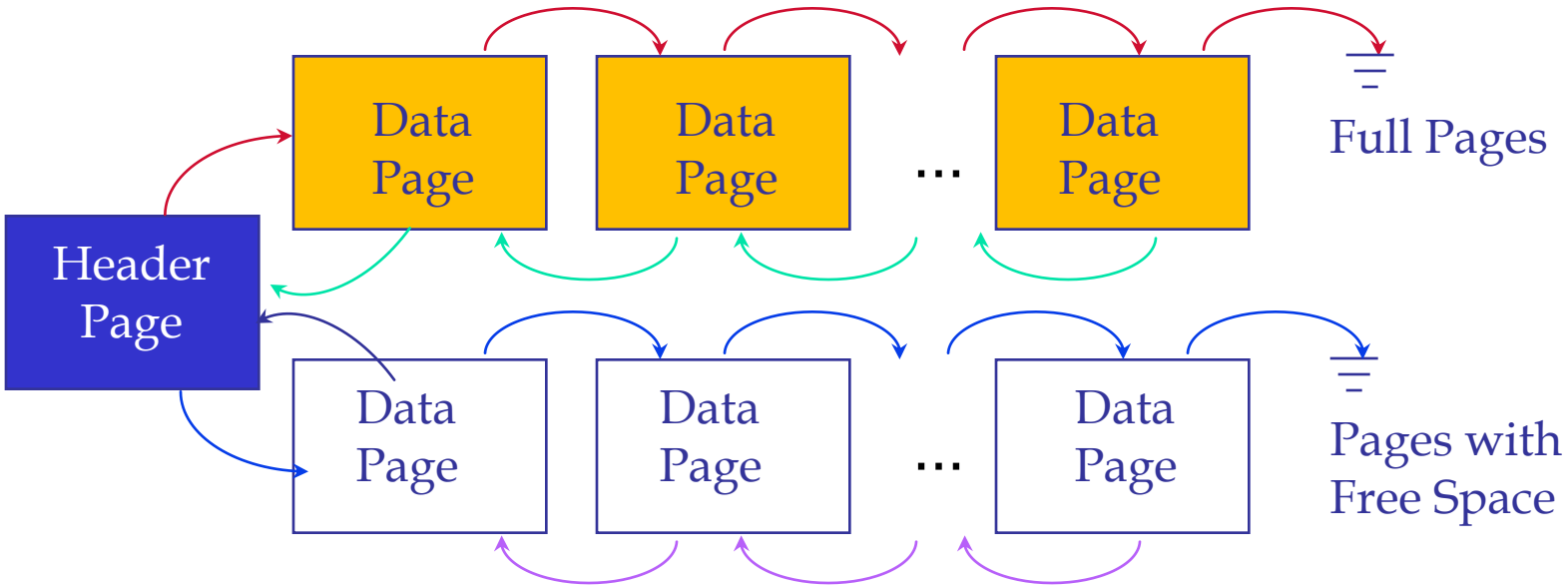


Physical Implementation

Disk-Resident Databases

File Organization – Relations (1)

- A table is stored in one or more files (a set of blocks)
- Each block may contain several records
- record id (rid) → block #
- Records are of fixed or variable length

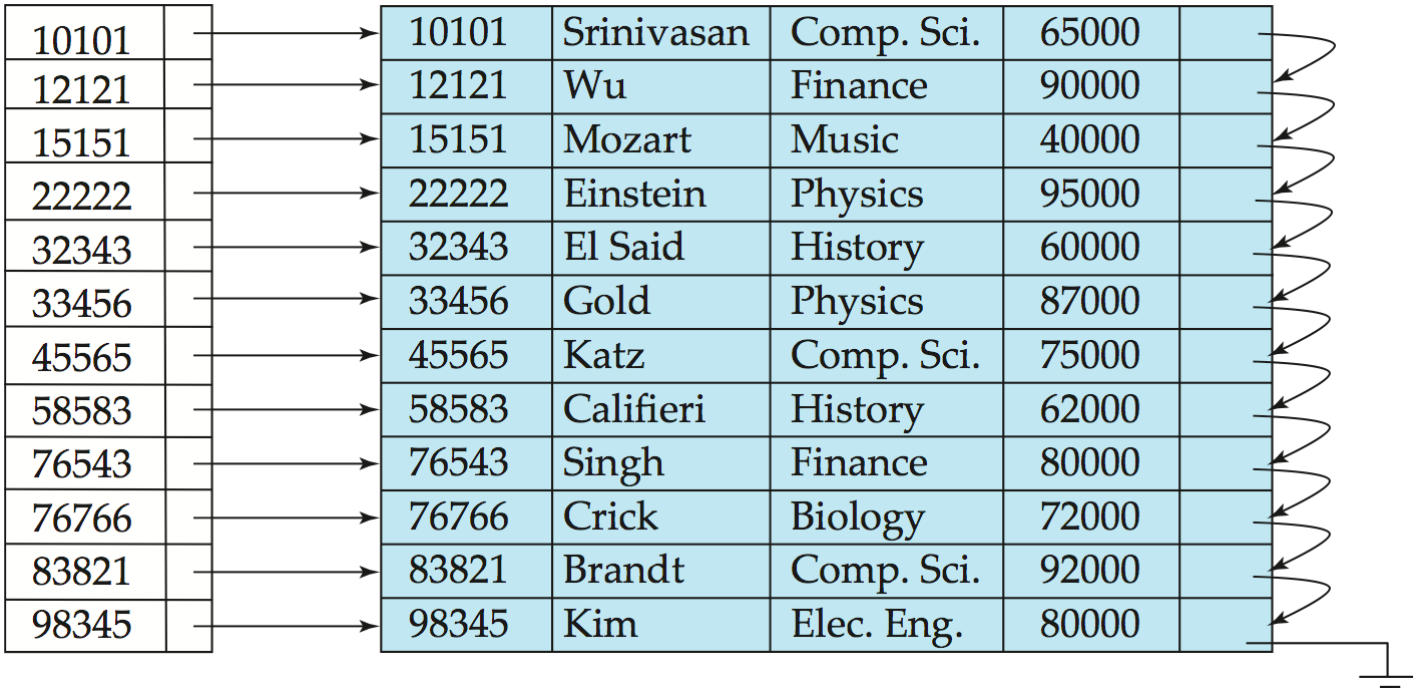



File Organization – Relations (2)

- Heap – a record is inserted at the end of the last block – if there is no space, get a new block
- Sequential – store records in sequential order, based on the value of some attribute (e.g. the primary key) of each record
- Hashing – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed

Indexing (1)

- Indexing mechanisms used to speed up access to desired data.
  - E.g. instructor id to instructors' detail data




Indexing (2)

- Search Key - attribute to set of attributes used to look up records in a file.
- An index file consists of records (called index entries) of the form

  (search-key, location) -- a table it self

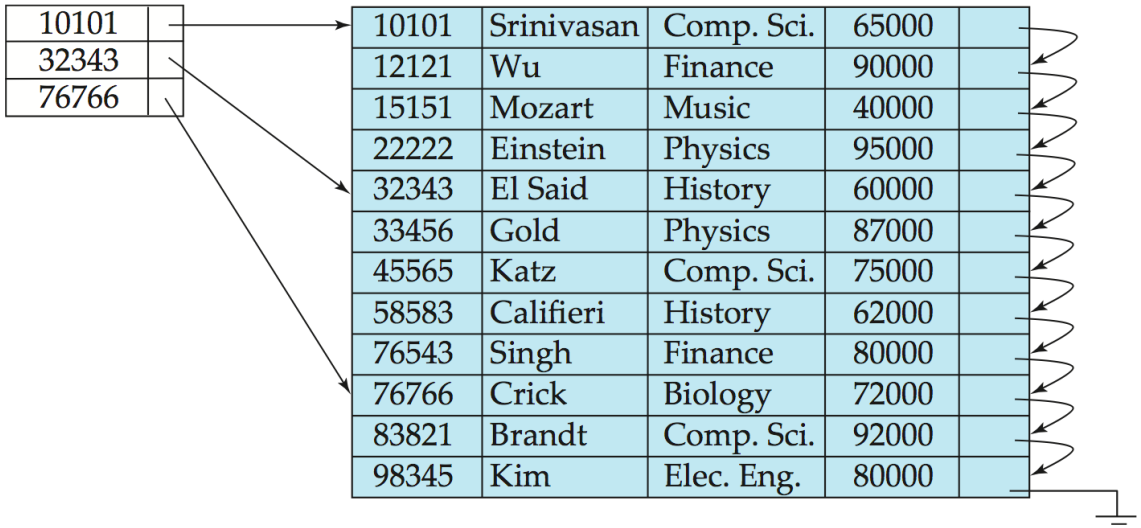
- Index files are typically much smaller than the original file – could fit in main memory
- Two basic kinds of indices:
  - Ordered indices: search keys are stored in sorted order
  - Hash indices: search keys are distributed uniformly across “buckets” using a “hash function”.

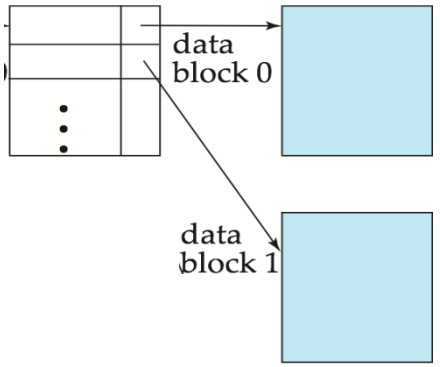

Indexing (3)

- Index Evaluation Metrics:
  - Access types supported efficiently:
    - records with a specified value in the attribute
    - or records with an attribute value falling in a specified range of values.
  - Access time
  - Insertion time
  - Deletion time
  - Space overhead

Indexing (4)

- Sparse/Dense Index
  - dense index: one entry per record – as previous slide
  - sparse index: one entry per block - file must be sorted on search key






Indexing (5)

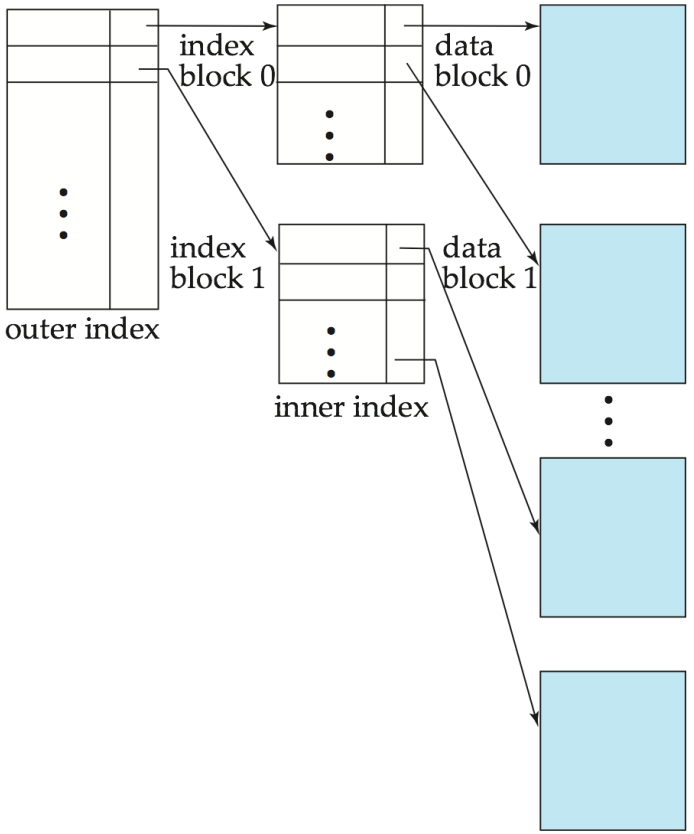
- How many sparse indexes can we have?
- How many dense indexes can we have?
- Other terms:
  - Clustered index (the sparse index, if exists)
  - Secondary indexes (more than one dense index)
- What if an index does not fit in main-memory?

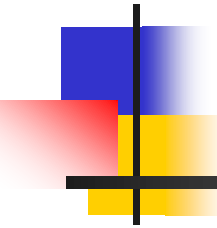

Indexing (6)

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Indexing (7)

- Similar to a tree
- B+ - trees have been developed for that purpose





Physical Implementation

Main-Memory Databases


MMDBs – Not a New Concept

- Main-memory data analysis tools are quite popular for several decades now
  - All the following are memory-based "BI" tools:
    - Excel, Lotus 1-2-3, VisiCalc, Cognos PowerPlay
- Main-memory DBs (MMDBs) are not new.
  - "Main Memory Database Systems: An Overview", H. Garcia-Molina & K. Salem, IEEE TKDE, Dec. 1992 (a must read)


MMDBs – MMDBs and DRDBs (1)

- Main Memory database system (MMDB)
  - Data resides permanently on main physical memory
  - Backup copy on disk
- Disk Resident database system (DRDB)
  - Data resides on disk
  - Data may be cached into memory for access
- Main difference is that in MMDB, the primary copy lives permanently in memory


MMDBs - MMDBs and DRDBs (2)

- Is it reasonable to assume that the entire database fits in memory?
  - Yes, for some applications (in the 80s)
    - 800 numbers, radar objects, securities trading
    - securities trading, radar tracking
  - Hot and cold data: accessed frequently/rarely
- What is the difference between a MMDB and a DRDB with a very large cache?
  - In DRDB, even if all data fits in memory, the structures and algorithms are designed for disk access


MMDBs – Differences (1)

- The differences in properties of main-memory and disk have important implications in:
  - Concurrency control
  - Commit processing
  - Access methods
  - Data representation
  - Query processing
  - Recovery
  - Performance


MMDBs – Differences (2)

- Concurrency Control
  - Transactions complete quicker than in DRDBs
    - serial execution – OK for short transactions; long ones?
    - serial execution removes the entire locking/swap overhead
  - If locking is used, smart implementations can be done
    - two lock bits per object
    - transactions set these bits
  - Multi-core, virtualization
  - Appropriate concurrency control mechanisms
    - e.g. multi version CC


MMDBs – Differences (2)

- Commit Processing
  - need for back up and log of transaction activity
  - the need for a stable log threatens to undermine the performance advantages of main-memory systems.
  - Solutions involve:
    - stable memory – periodically flush contents to disk
    - pre-committing – write a small amount to disk
    - group commits
    - data redundancy by replicating partitions to different nodes


MMDBs – Differences (3)

- Access Methods
  - Different access methods (i.e. indices) than disk. Various forms of hashing, trees, pointer following. Data need not be stored with index.
  - The use of pointers suggests perhaps the simplest way to provide an index, which is simply to invert the relation on the indexed field. In a main memory DB, the inverted “relation” can simply be a list of tuple pointers in sorted order. Such indexes are space efficient and fast for range and exact-match queries.


MMDBs – Differences (4)

- Data Representation
  - How relations are represented in main-memory using pointers? (e.g. relational tuples can be represented as a set of pointers to data values)
- Query Processing
  - SQL statements and relational operators must be mapped to the underlying data representation of the relations. Projection, selection and join algorithms depend dramatically on the underlying representation. Sometimes, joins are already materialized through pointer following (tuples to values to tuples).


MMDBs – Differences (5)

- Recovery
  - Checkpointing + disk resident copy
  - Only system transactions hit the disk – application transactions only use main-memory DB
- Performance
  - Metrics are different compared to DRDBs (where I/O operations determine the performance of an algorithm)
  - Metrics can involve checkpointing and backups


MMDBs – Differences (6)

- Application Programming Interface & Protection
  - In DRDB, data exchange (r/w) is done through private buffers and system calls – in MMDB give actual memory location
- Data Clustering and Migration
  - IN DRDB, cluster data accessed together, close to each other
  - How do you cluster things together when you migrate to disk?
    - vacuum cleaning
    - serialization/deserialization


Main-Memory Systems (in late 80s)

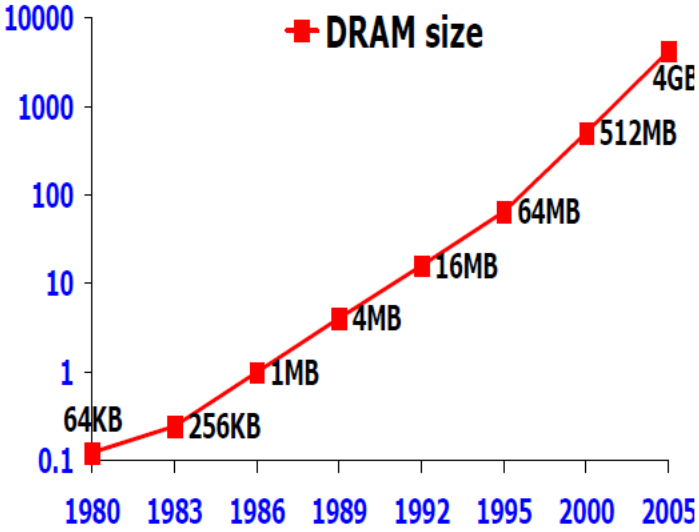
	Concurrency	Committ Processing	Data Representation	Access Methods	Query Processing	Recovery
MM-DBMS	two-pahse locking of relations	stable log tail by segment	self-contained segments, heap per segment, extensive pointer use	hashing, T-trees, pointers to values	merge, nested-loop, joins	segments recovered on demand, recovery processor
MARS	two-phase locking of relations	stable shadow memory, log tail				recovery processor, fuzzy checkpoints
HALO		in hardware, nearly transparent				physical, word-level log
OBE			extensive use of pointers	inverted indexes	nested loop-join, on-the-fly-index creation, optimization focuses on processor costs	
TPK	serial transaction execution	group committ, precommitt	arrays			two memory resident databases, fuzzy checkpoints
System M	two-phase locking, minimize concurrency	several alternatives	self-contained segments, heap per segment			various checkpointing, logging options
Fast Path	VERIFY/CHANGE for hot spots	group committ				

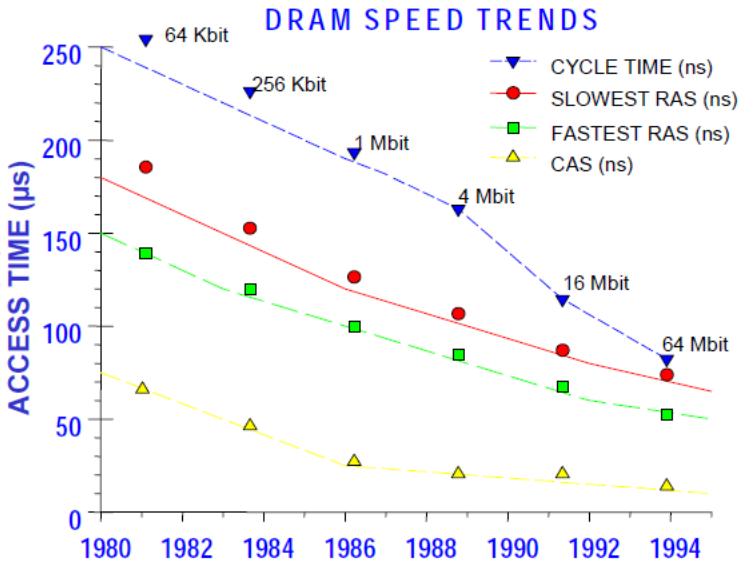

Main Memory DBs - Rebirth

- 2008: H-Store → VoltDB
  - "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System", VLDB 2008
  - "OLTP Through the Looking Glass, and What We Found There", SIGMOD 2008
- 2011: P*-Times → SAP Hana
  - Claim: Both for OLTP and OLAP applications
  - "A Course in In-Memory Data Management: The Inner Mechanics of In-Memory", Hasso Plattner, Springer, 2013
- 2005: TimesTen → Oracle Exadata
- 2012+: MemSQL, Microsoft, …

Main-Memory DBs – Hardware Trends

- 64-bit operating systems
- Multi-cores






OLTP Through the Looking Glass (1)

- Advances in modern processors, memories, and networks mean that today's computers are vastly different from those of 30 years ago, such that many OLTP databases will now fit in main memory, and most OLTP transactions can be processed in milliseconds or less.
- Yet database architecture has changed little.


OLTP Through the Looking Glass (2)

- Shore, TPC-C benchmarking
- Four major components, gradually removed:
  - **Logging.** Assembling log records and tracking down all changes in database structures slows performance. Logging may not be necessary if recoverability is not a requirement or if recoverability is provided through other means (e.g., other sites on the network).
  - **Locking.** Traditional two-phase locking poses a sizeable overhead since all accesses to database structures are governed by a separate entity, the Lock Manager.


OLTP Through the Looking Glass (3)

- Four major components, gradually removed (cont.):
  - **Latching.** In a multi-threaded database, many data structures have to be latched before they can be accessed. Removing this feature and going to a single-threaded approach has a noticeable performance impact.
  - **Buffer management.** A main memory database system does not need to access pages through a buffer pool, eliminating a level of indirection on every record access.
- From about 640 tps to 12700 tps

OLTP Through the Looking Glass (4)

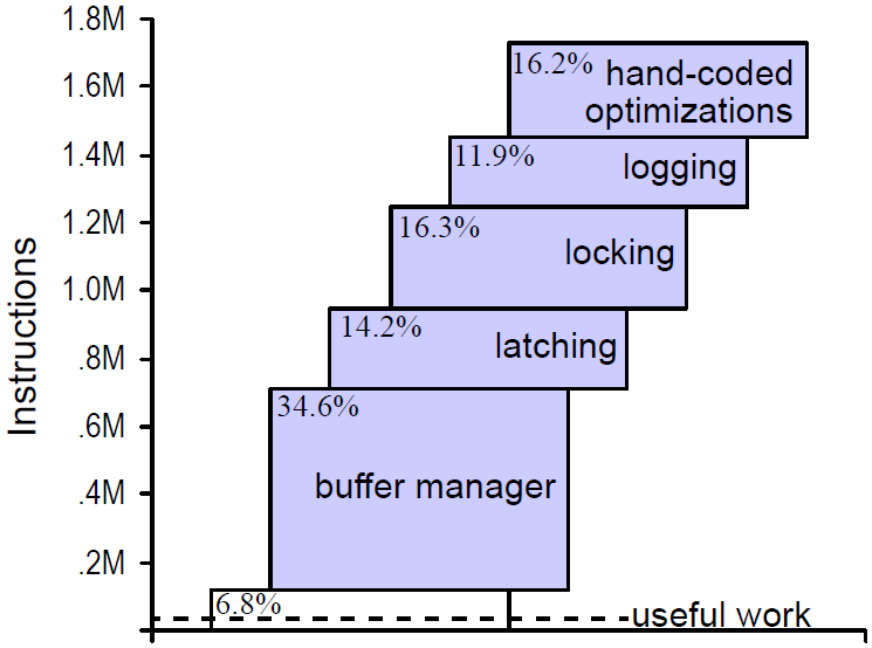


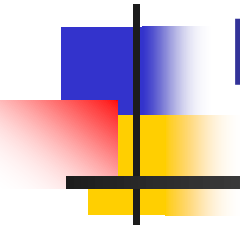
Figure 1. Breakdown of instruction count for various DBMS components for the New Order transaction from TPC-C. The top of the bar-graph is the original Shore performance with a main memory resident database and no thread contention. The bottom dashed line is the useful work, measured by executing the transaction on a no-overhead kernel.


OLTP Through the Looking Glass (5)

- Trends in OLTP
  - Cluster computing
  - Memory Resident Databases
  - Single Threading in OLTP Systems
  - High Availability vs Logging
  - Transaction Variants


Main-Memory Databases

- P*time → SAP Hana
- TimesTen → Oracle
- MemSQL
- VoltDB
- Microsoft Hekaton



Physical Implementation

Column-Oriented Databases


Column-Oriented – Not a New Concept (1)

- Transposed files (late 70s):

Suppose A is partitioned into k nonempty sets A_j $(j = 1, \ldots, k)$ on which subfiles SF_j are defined. Thus record R_i of the original file is partitioned into k disjoint subrecords R_{ij}. Each subfile is implemented as a *nonsequential file* [15] (i.e., a file where records are stored chronologically) with the property that the ith subrecord stored in subfile SF_j is R_{ij}. Thus R_i is reconstructed by reading the ith subrecord of all k subfiles. This collection of subfiles is called a *partitioned transposed file*. If one attribute is assigned to each subfile, the resulting structure is *fully transposed*. Had A been subdivided into k nonempty overlapping sets A_j where $\bigcup_{j=1}^{k} A_j = A$ and $k > 1$, the resulting collection of subfiles is called a *clustered transposed file* (see Figure 1).

Column-Oriented – Not a New Concept (2)

- Transposed files (late 70s):

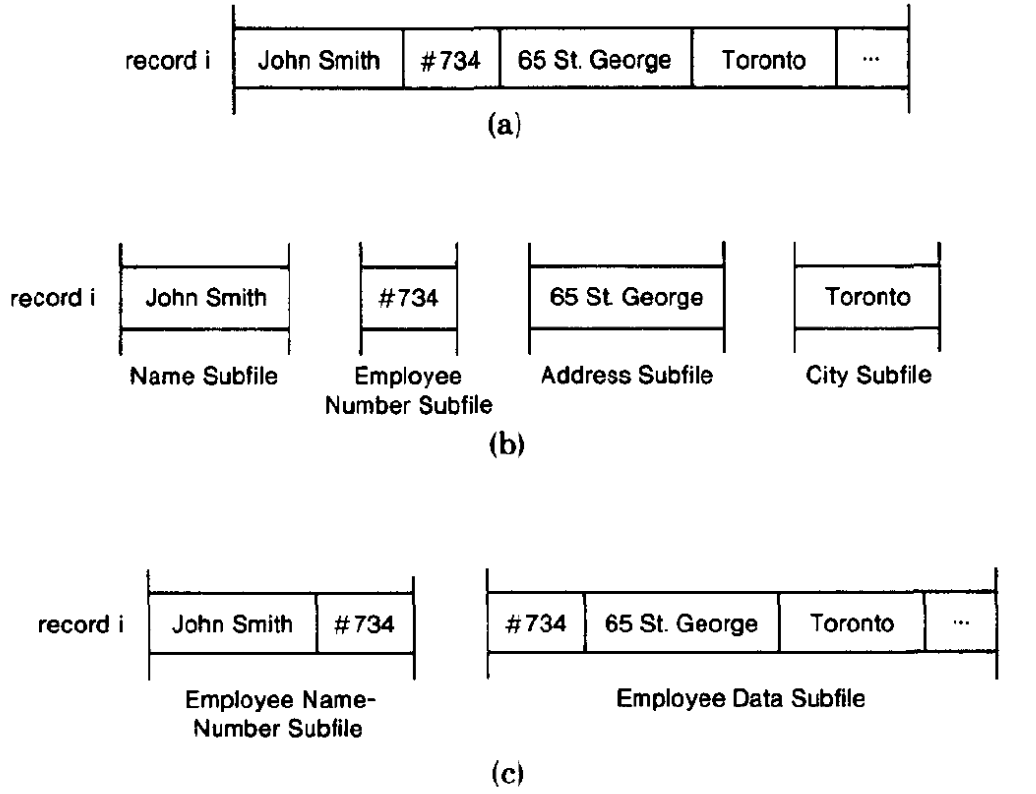
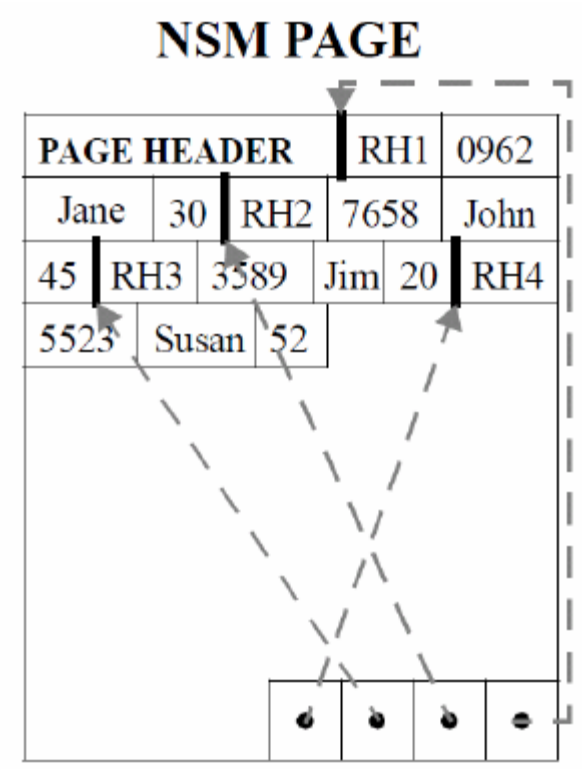
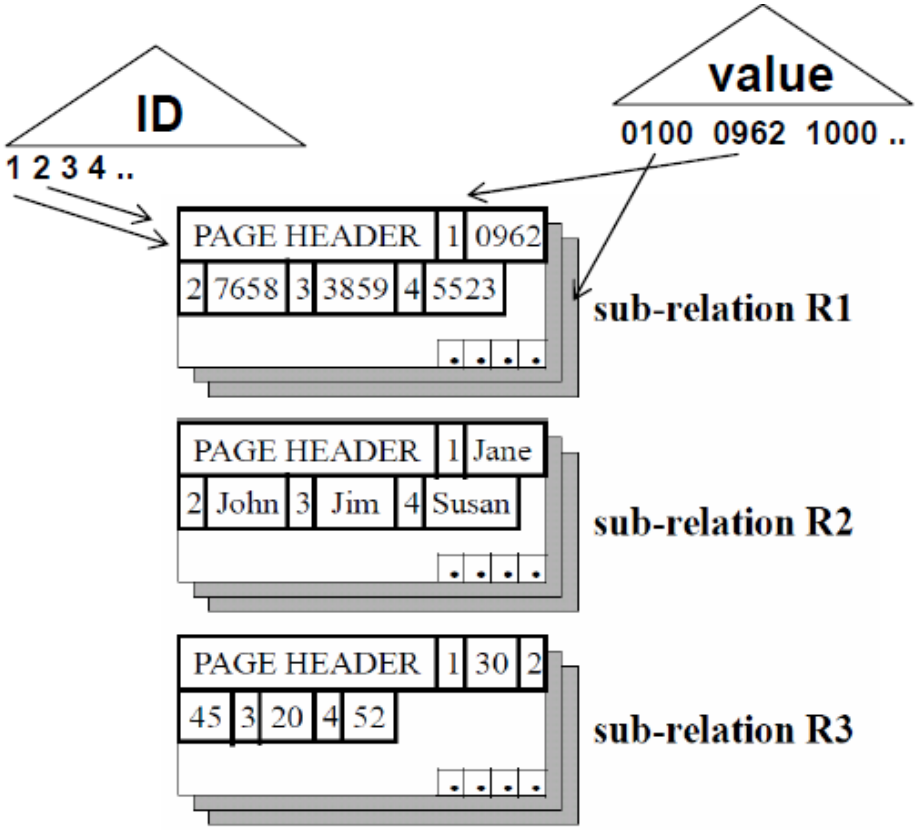


Fig. 1. (a) Nonsequential file layout. (b) Partitioned transposed file layout. (c) Clustered transposed file layout.

Column-Oriented – Not a New Concept (3)

- A Decomposition Storage Model (DSM), 1985






Column-Oriented – Not a New Concept (4)

- A Decomposition Storage Model (DSM), 1985
  - Proposed as an alternative to NSM
  - 2 indexes: clustered on ID, non-clustered on value
  - Speeds up queries projecting few columns
  - Requires more storage
- Sybase IQ (1994)
- PAX (2001)
- MonetDB (2002)

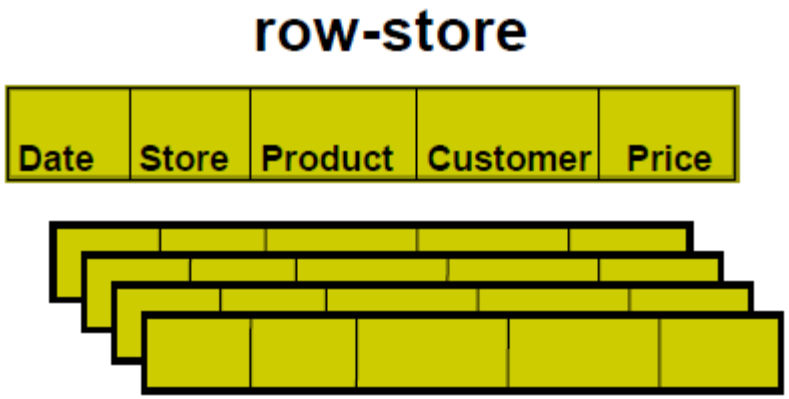

Column-Oriented Databases – Reading

- "One size fits all: an idea whose time has come and gone", Stonebraker et al., ICDE 2005
  - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.9136&rep=rep1&type=pdf
- "C-Store", Stonebraker et al., VLDB 2005
  - http://people.csail.mit.edu/tdanford/6830papers/stonebraker-cstore.pdf
- "One Size Fits All? – Part 2: Benchmarking Results", Stonebraker et al., CIDR 2007
- "Tutorial: Column-Oriented Database Systems", Harizopoulos et al., VLDB 2009
  - http://cs-www.cs.yale.edu/homes/dna/talks/Column_Store_Tutorial_VLDB09.pdf


Column-Oriented Databases - Systems

- Rebirth of column-oriented storage:
  - C-Store (MIT) → Vertica
  - MonetDB (CWI)
  - SAP
  - Oracle

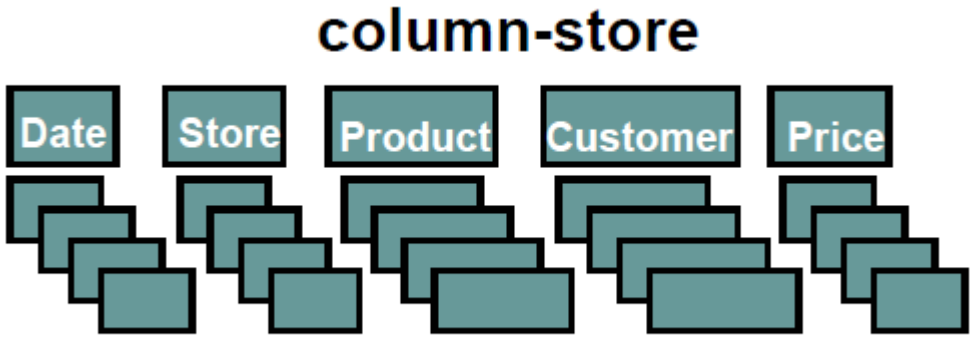
Column-Oriented Databases – Main Idea (1)



Source: Column-Oriented Databases Tutorial, Harizopoulos et al., VLDB 2009

- Good for: add/modify records
- Bad for: read unnecessary data during queries

Column-Oriented Databases – Main Idea (2)



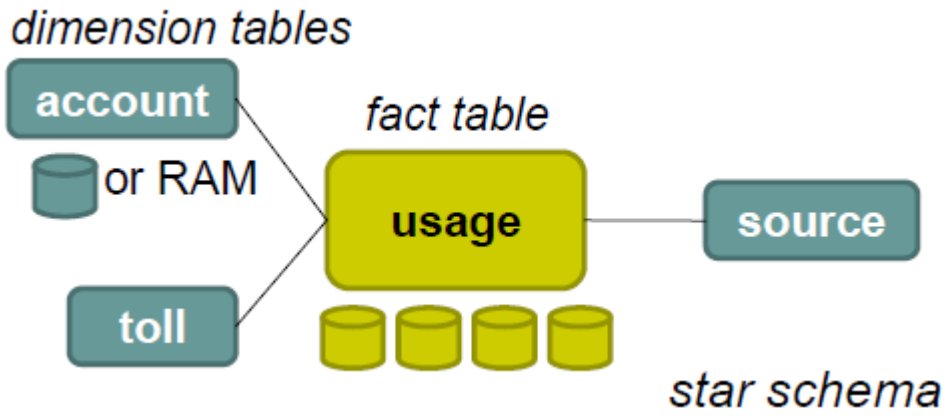
Source: Column-Oriented Databases Tutorial, Harizopoulos et al., VLDB 2009

- Good for: reads in only relevant data (queries)
- Bad for: tuple updates – too many writes

➔ *suitable for read-mostly, read-intensive, large data repositories*

Column-Oriented DBs – DW, Example (1)

- Example: A Telco data warehouse



Source: "One Size Fits All? - Part 2: Benchmarking Results", Stonebraker et al. CIDR 2007

Column-Oriented DBs – DW, Example (2)

- Benchmarking on 5 queries

```
QUERY 2
SELECT account.account_number,
sum (usage.toll_airtime),
sum (usage.toll_price)
FROM usage, toll, source, account
WHERE usage.toll_id = toll.toll_id
AND usage.source_id = source.source_id
AND usage.account_id = account.account_id
AND toll.type_ind in ('AE'. 'AA')
AND usage.toll_price > 0
AND source.type != 'CIBER'
AND toll.rating_method = 'IS'
AND usage.invoice_date = 20051013
GROUP BY account.account_number
```

	Col-store	Row-store
Query1	2.06	300
Query2	2.20	300
Query3	0.09	300
Query4	5.24	300
Query5	2.88	300

Source: Column-Oriented Databases Tutorial, Harizopoulos et al., VLDB 2009


Column-Oriented DBs – DW, Example (3)

- Read Efficiency
  - Row-stores
    - read pages containing entire rows
    - one row = 212 columns
  - Column-stores
    - read only columns needed (in this example: 7)
    - caveats:
      - “select * ” not any faster
      - requires clever disk prefetching
      - requires clever tuple reconstruction

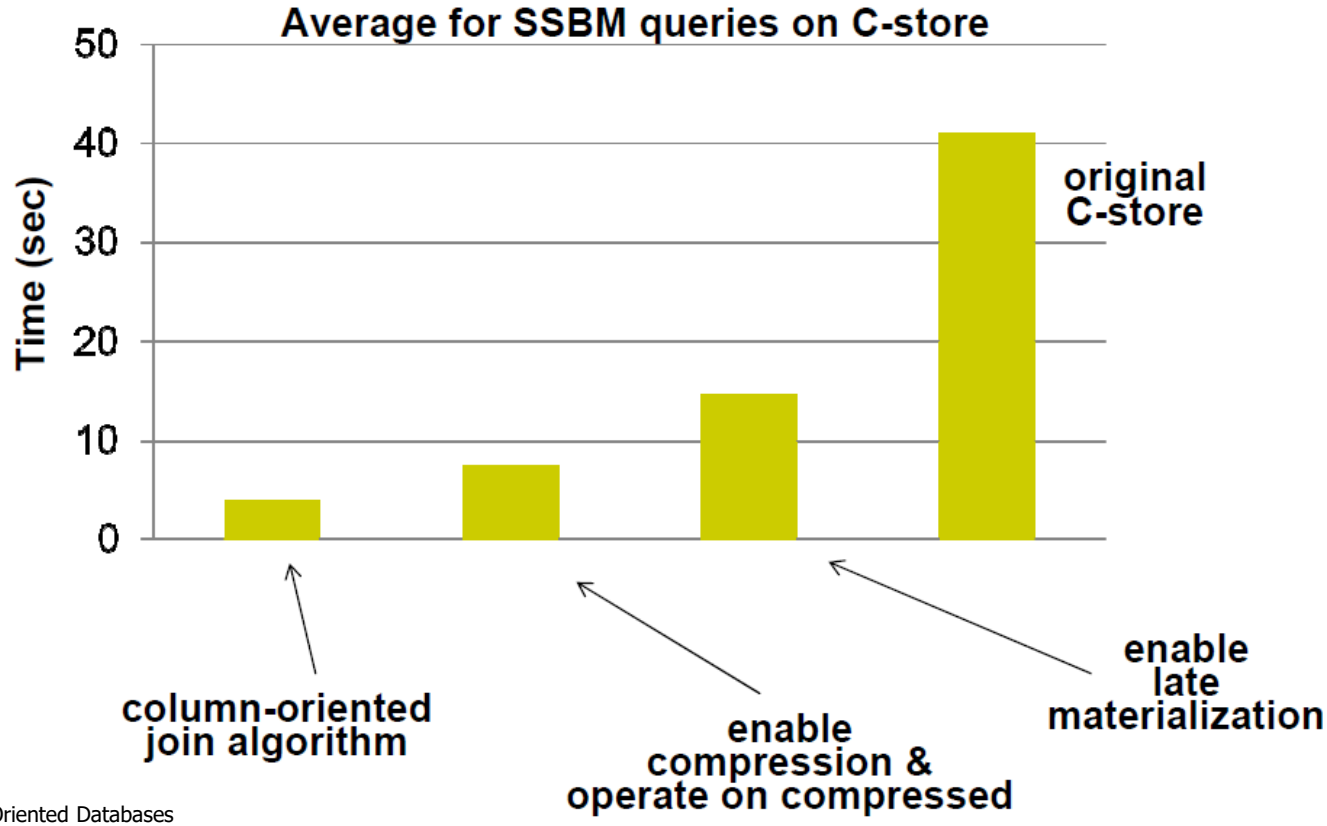

Column-Oriented DBs – DW, Example (4)

- Better compression
  - Columns compress better than rows
    - Typical row-store compression ratio 1 : 3
    - Column-store 1 : 10
  - Rows contain values from different domains
    - more entropy, difficult to dense-pack
  - Columns exhibit significantly less entropy
    - Examples:
      - Male, Female, Female, Female, Male
      - 1998, 1998, 1999, 1999, 1999, 2000
  - Problem: CPU cost (use lightweight compression)


Column-Oriented DBs – DW, Example (5)

- Sorting and Indexing Efficiency
  - Compression and dense-packing free up space
    - Use multiple overlapping column collections
    - Sorted columns compress better
    - Range queries are faster
    - Use sparse clustered indexes

Column-Oriented DBs – C-Store + Opts



Source: Column-Oriented Databases Tutorial, Harizopoulos et al., VLDB 2009

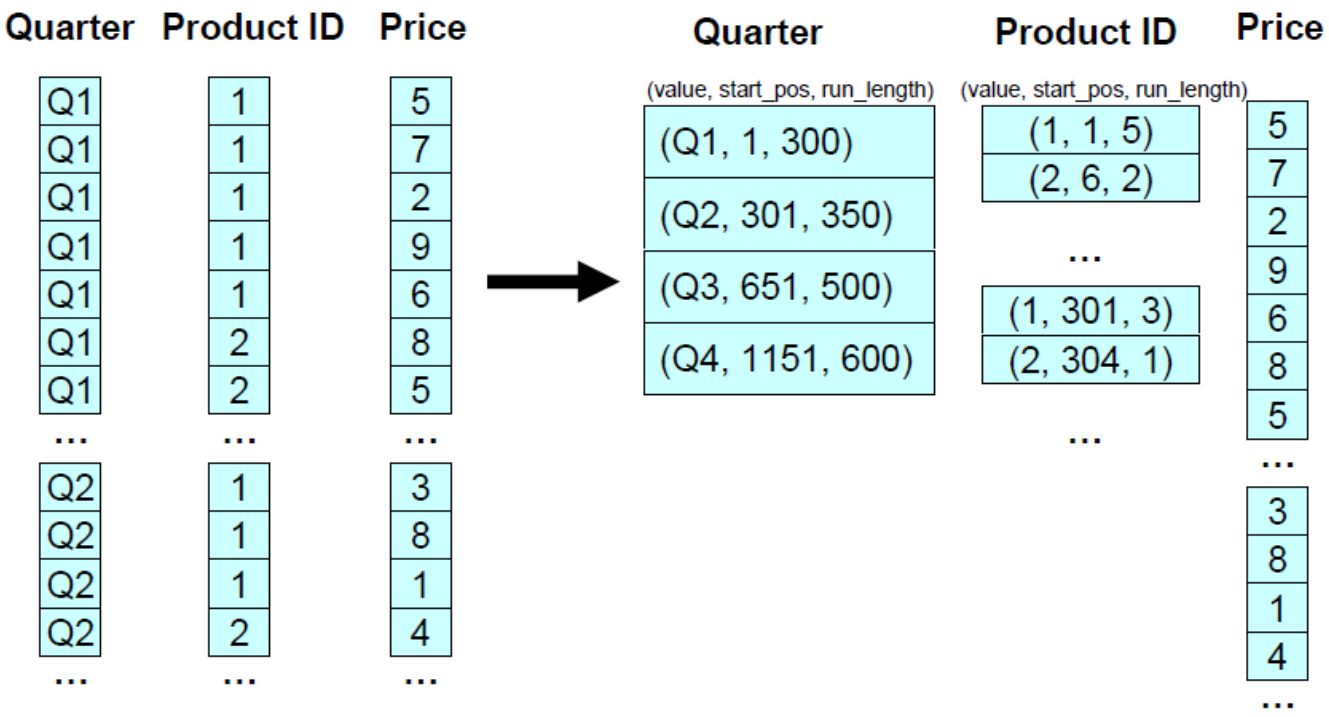

Column-Oriented DBs – Compression (1)

- Trades I/O for CPU
- Increased column-store opportunities:
  - Higher data value locality in column stores
  - Techniques such as run length encoding far more useful
  - Can use extra space to store multiple copies of data in different sort orders

Column-Oriented DBs – Compression (2)

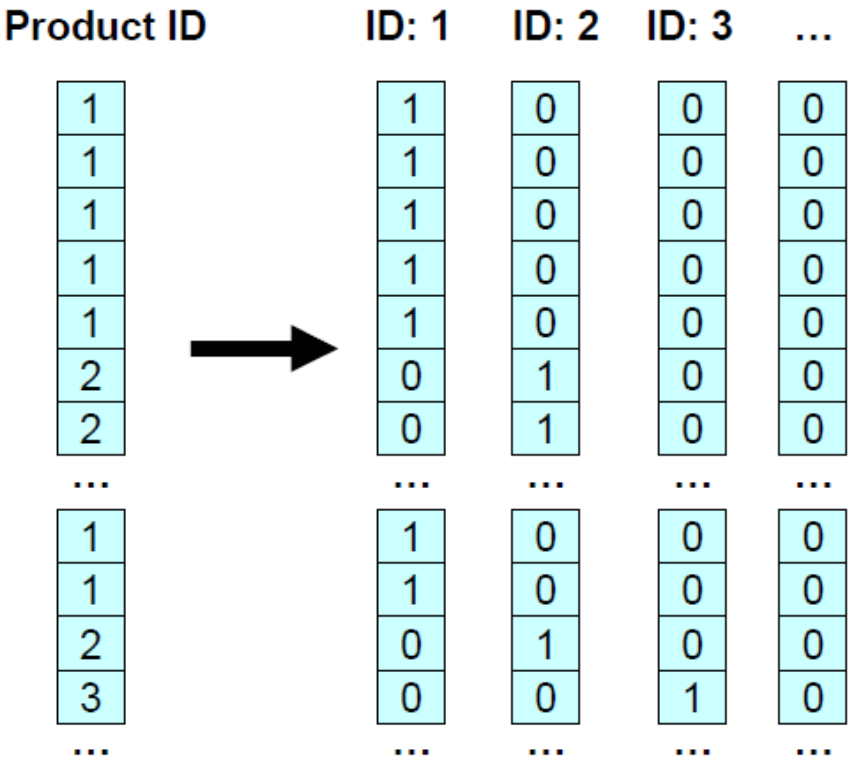
- Run-Length Encoding (RLE)

Source: Column-Oriented Databases Tutorial, Harizopoulos et al., VLDB 2009



Column-Oriented DBs – Compression (3)

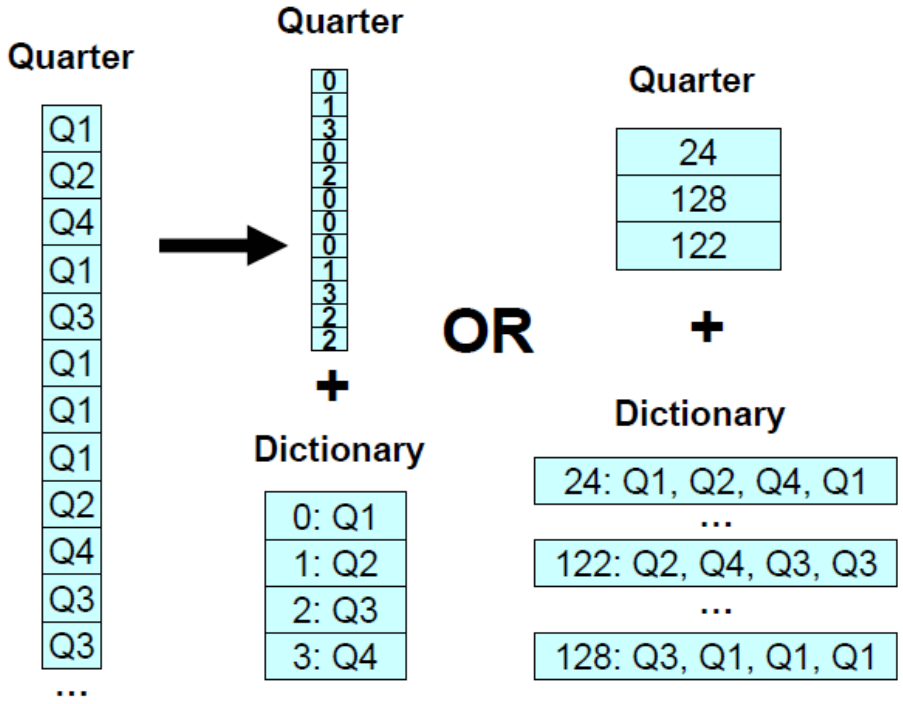
- Bit-Vector Encoding



Source: Column-Oriented Databases Tutorial, Harizopoulos et al., VLDB 2009

Column-Oriented DBs – Compression (4)

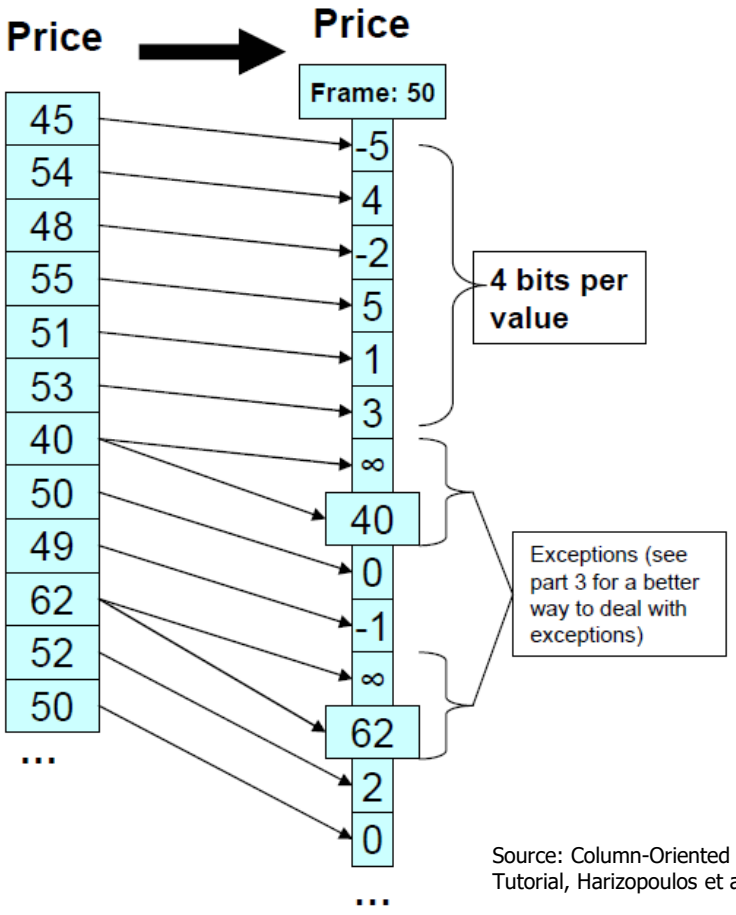
- Dictionary Encoding
  - For each unique value create dictionary entry
  - Dictionary can be per-block or per-column
  - Column-stores have the advantage that dictionary entries may encode multiple values at once



Source: Column-Oriented Databases Tutorial, Harizopoulos et al., VLDB 2009

Column-Oriented DBs – Compression (5)

- Frame of Reference
  - Encodes values as b bit offset from chosen frame of reference
  - Special escape code (e.g. all bits set to 1 indicates a difference larger than can be stored in b bits
  - After escape code, original (uncompressed) value is written



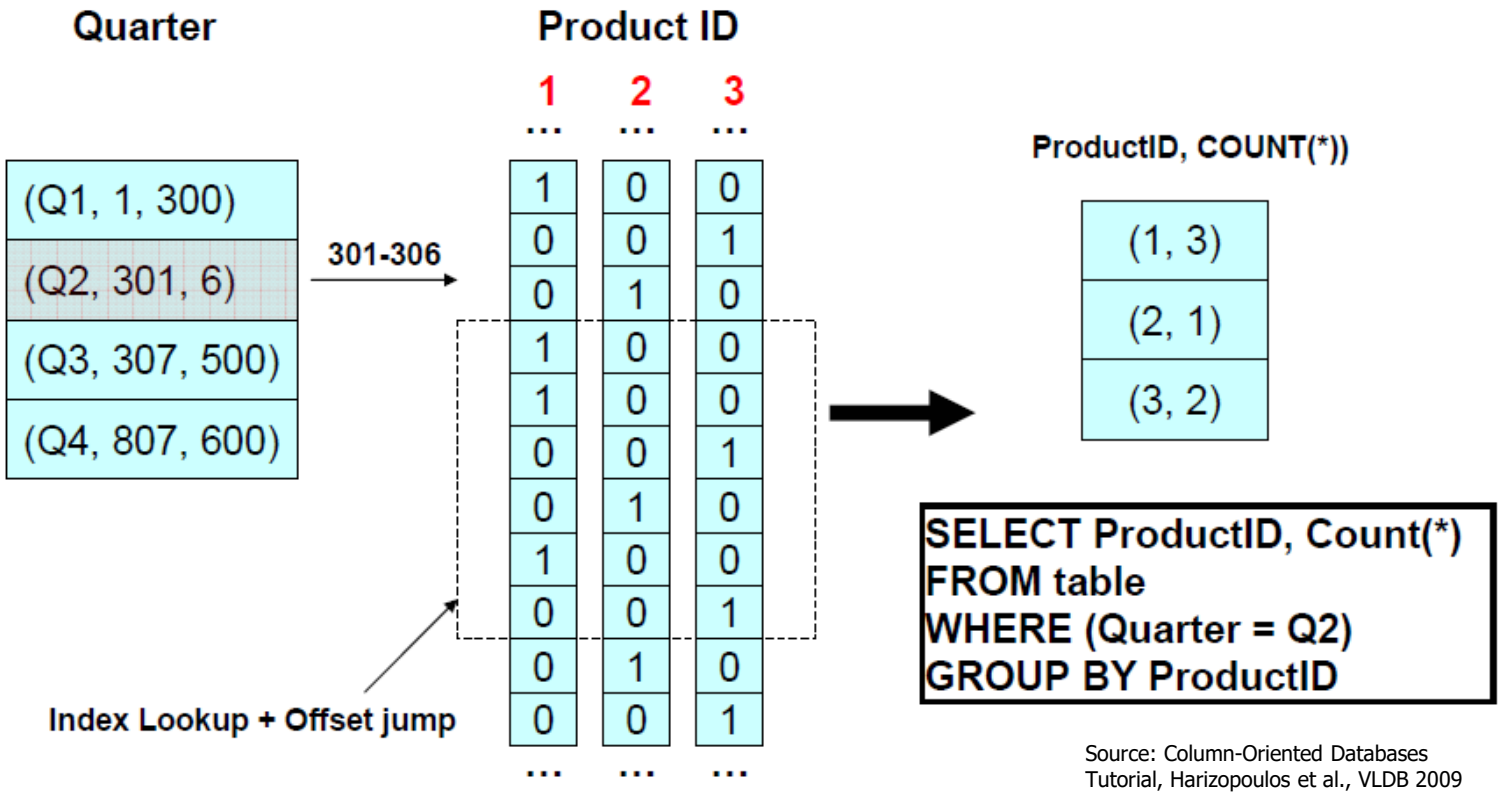
Source: Column-Oriented Databases Tutorial, Harizopoulos et al., VLDB 2009


Column-Oriented DBs – Compression (6)

- Other light-weight encoding methods
  - Differential
- Heavy-weight encoding techniques
  - BZIP
  - ZLIB
  - LZO

Column-Oriented DBs – Compression (7)

- Operating directly on compressed data



Source: Column-Oriented Databases Tutorial, Harizopoulos et al., VLDB 2009


Column-Oriented DBs – Query Processing

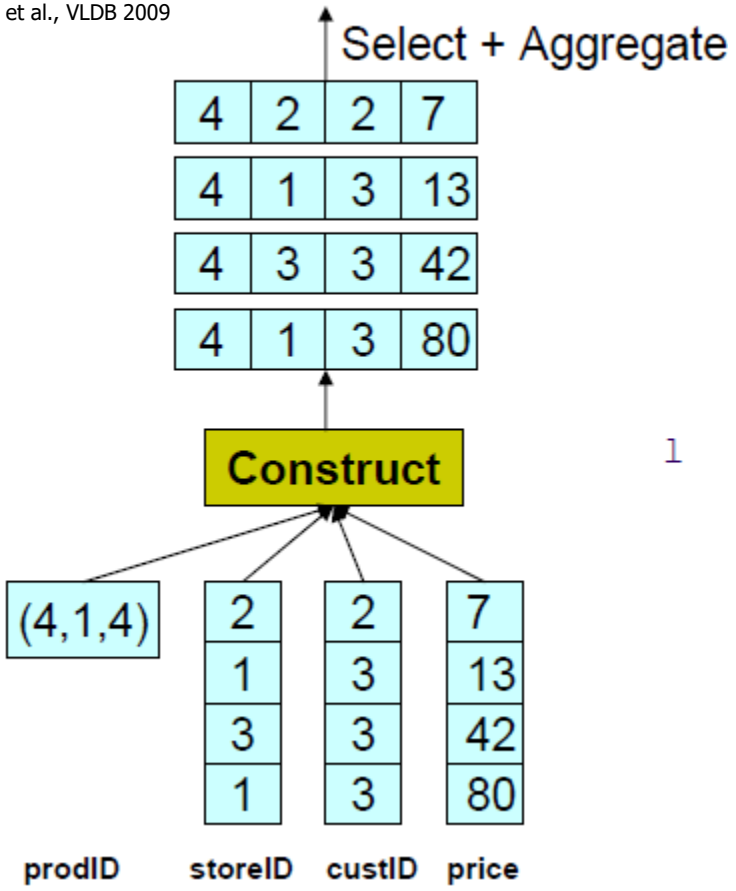
- Where should column projection operators be placed in a query plan?
  - Row-store: Column projection involves removing unneeded columns from tuples. Generally done as early as possible
  - Column-store: Operation is almost completely opposite from a row-store. Column projection involves reading needed columns from storage and extracting values for a listed set of tuples. This process is called “materialization”.
    - Early materialization: project columns at beginning of query plan
    - Late materialization: wait as long as possible for projecting columns

Column-Oriented DBs – Query Processing

```
QUERY:

SELECT custID,SUM(price)
FROM table
WHERE (prodID = 4) AND
      (storeID = 1) AND
GROUP BY custID
```

Source: Column-Oriented Databases Tutorial, Harizopoulos et al., VLDB 2009



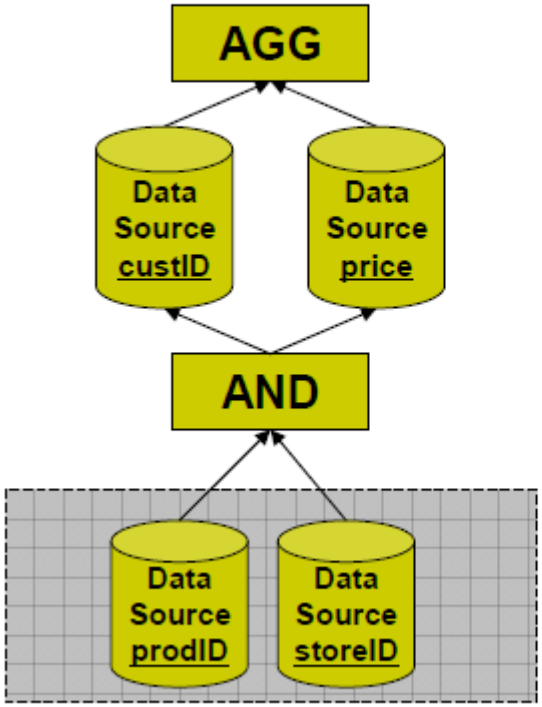
- Approach #1: Early Tuple Construction
  - Need to construct ALL tuples
  - Need to decompress data
  - Poor memory bandwidth utilization

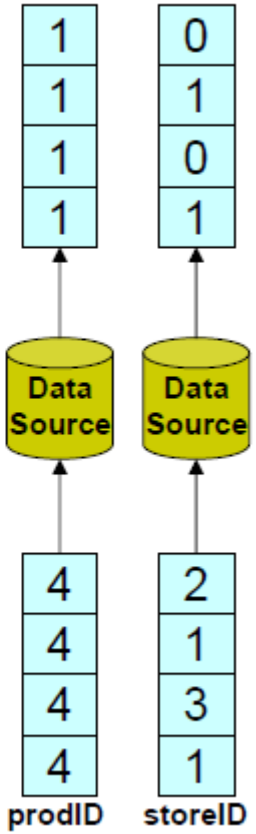
Column-Oriented DBs – Query Processing

```
QUERY:

SELECT custID,SUM(price)
FROM table
WHERE (prodID = 4) AND
      (storeID = 1) AND
GROUP BY custID
```

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80



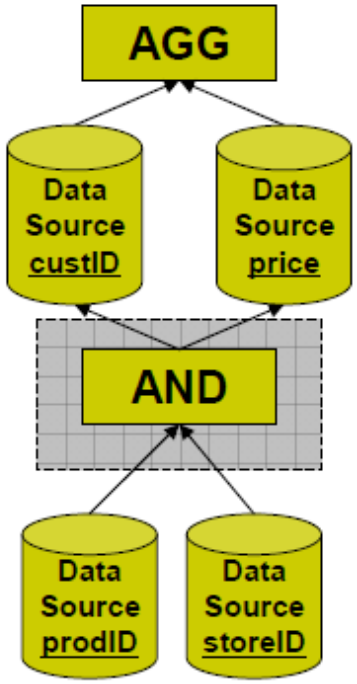


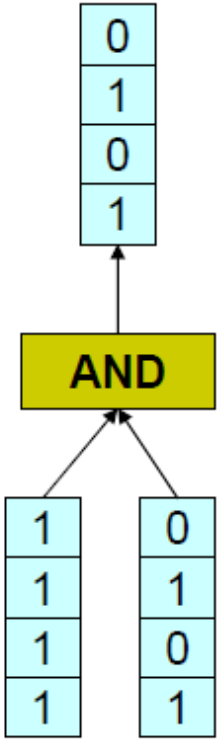
Column-Oriented DBs – Query Processing

```
QUERY:

SELECT custID,SUM(price)
FROM table
WHERE (prodID = 4) AND
      (storeID = 1) AND
GROUP BY custID
```

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

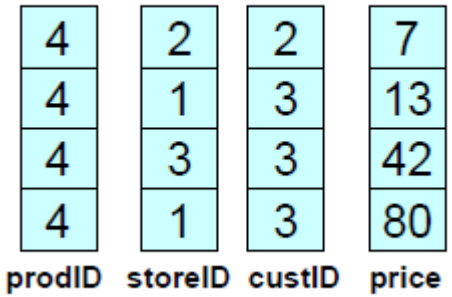


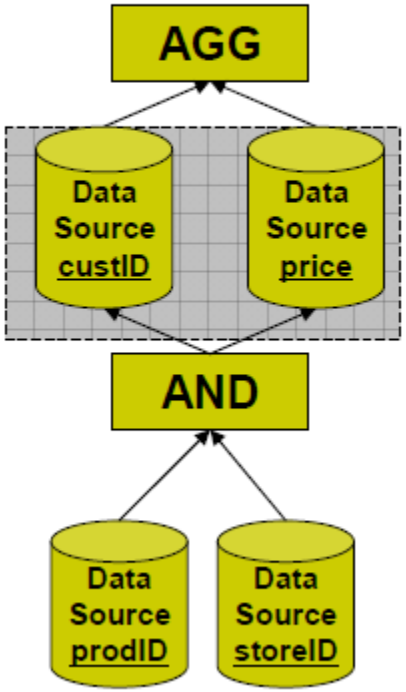


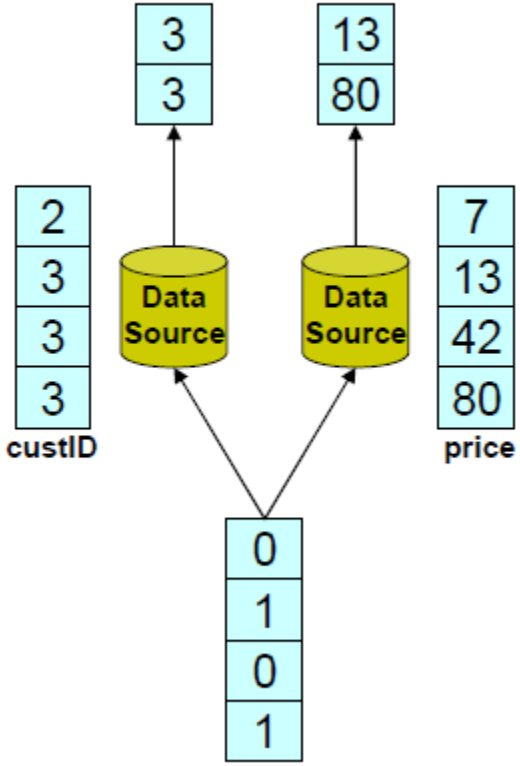
Column-Oriented DBs – Query Processing

```
QUERY:

SELECT custID,SUM(price)
FROM table
WHERE (prodID = 4) AND
      (storeID = 1) AND
GROUP BY custID
```





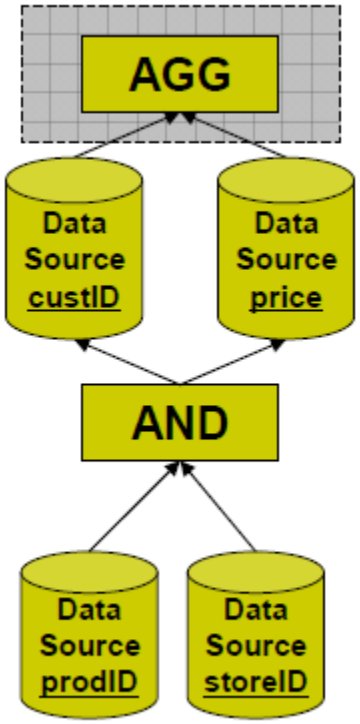


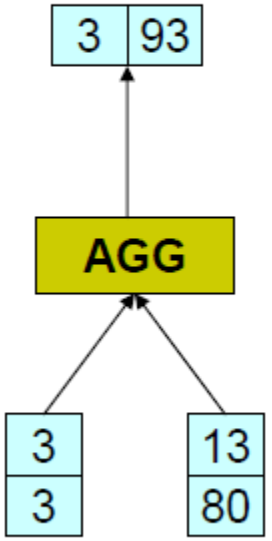
Column-Oriented DBs – Query Processing

```
QUERY:

SELECT custID,SUM(price)
FROM table
WHERE (prodID = 4) AND
      (storeID = 1) AND
GROUP BY custID
```

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80



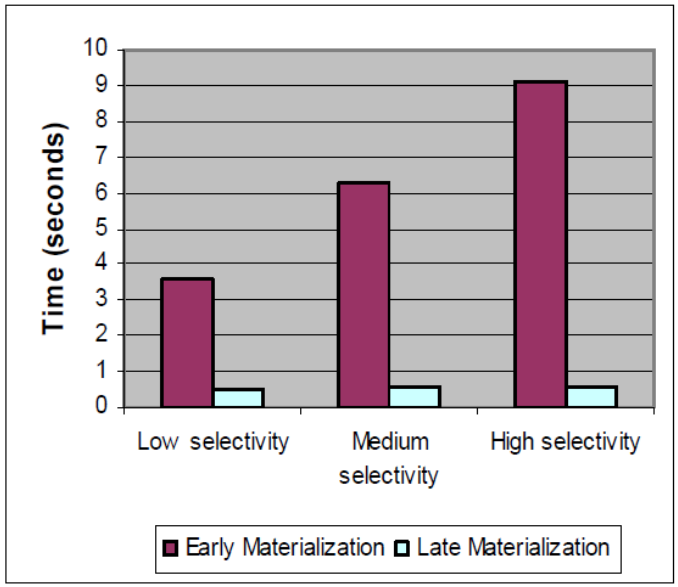


Column-Oriented DBs – Query Processing

- Late materialization is a win for plans with no joins

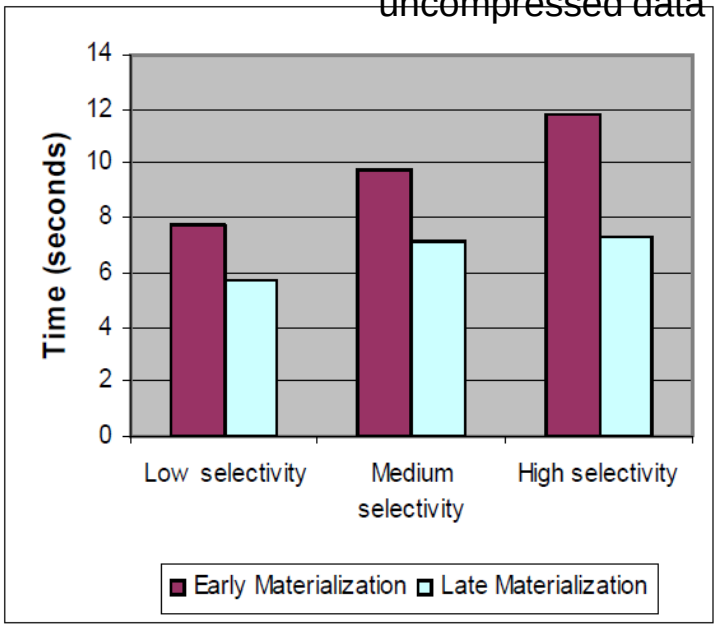
QUERY:

```
SELECT C1, SUM(C2)
FROM table
WHERE (C1 < CONST) AND
      (C2 < CONST)
GROUP BY C1
```



compressed data

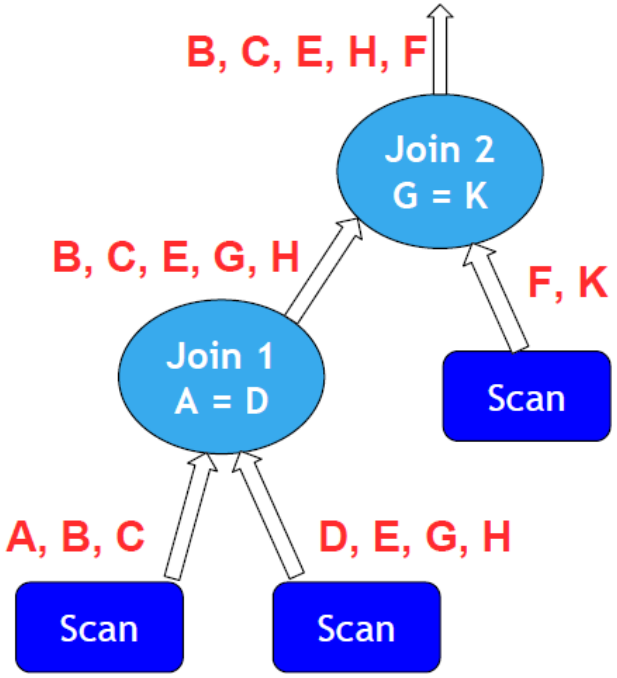
uncompressed data



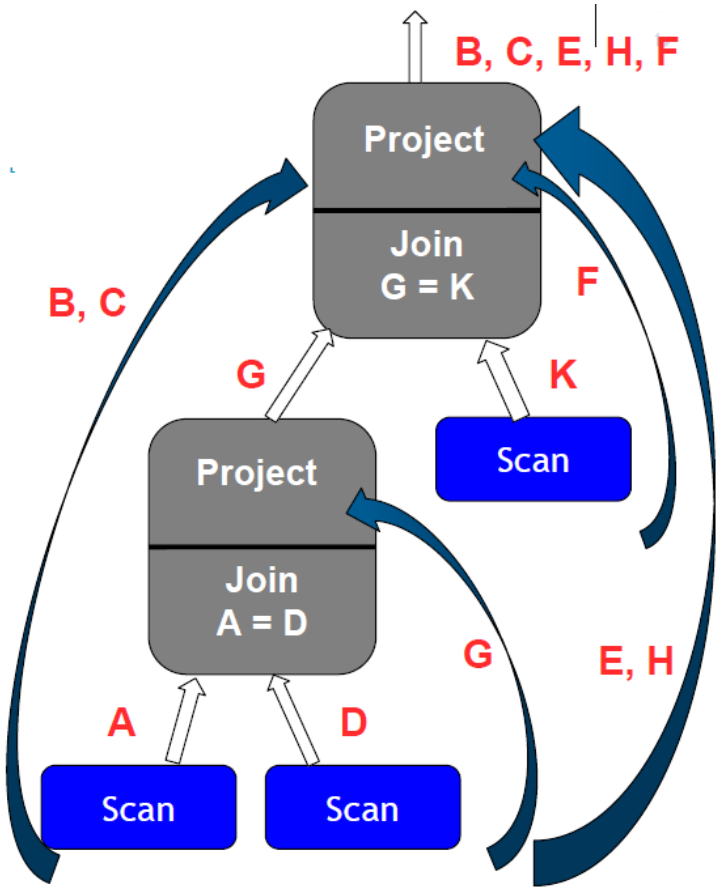
Source: Column-Oriented Databases Tutorial, Harizopoulos et al., VLDB 2009

Column-Oriented DBs – Query Processing

```
Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K
```



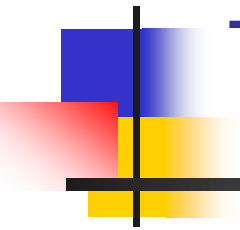
Early materialization



Late materialization


Column-Oriented DBs – Other Issues

- Compression techniques
- Operating directly on compressed data
- Novel/specific join algoriths
  - Invisible join (between fact and dimension tables)
  - jive/flash join
  - radix cluster/decluster join
- In-memory techniques and utilization
- Novel architectures (hardware) for DB processing



Transactions

Transactions, Definition (1)

- A transaction is a unit of program execution – *a single logical function* – that accesses and possibly updates various data items.
- Example: transfer 50€ from account A to B:

1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)


Transactions, Definition (2)

- Hardware and software failures happen.
- Transactions should be allowed to run concurrently for performance reasons.

Failures, Concept

- What will happen if system fails between steps 3 and 4?
  - Account A will have 50€ less than what it should (database will be in an inconsistent state.)
- Transactions should always either complete (commit) or undo all actions (rollback).
- *Transaction-management component* ensures that the database remains in a consistent state despite system and transaction failures.

Concurrency, Concept

- What will happen if another transaction reads A and B, between steps 3 and 4?
  - the sum of A and B is wrong
- Transactions should *appear* to execute sequentially, although they run concurrently.
- *Concurrency-control manager* controls the interaction among the concurrent transactions, to ensure the consistency of the database.

T1

1. read(A)
2. A := A - 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

T2

read(A); read(B); output(A+B)


ACID Properties (1)

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
  - When the user is notified that the transfer has been completed, no software or hardware failures can change this.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
  - The database remains is in a consistent state when a transaction begins and when it completes. In the mean time it may be in an inconsistent state.

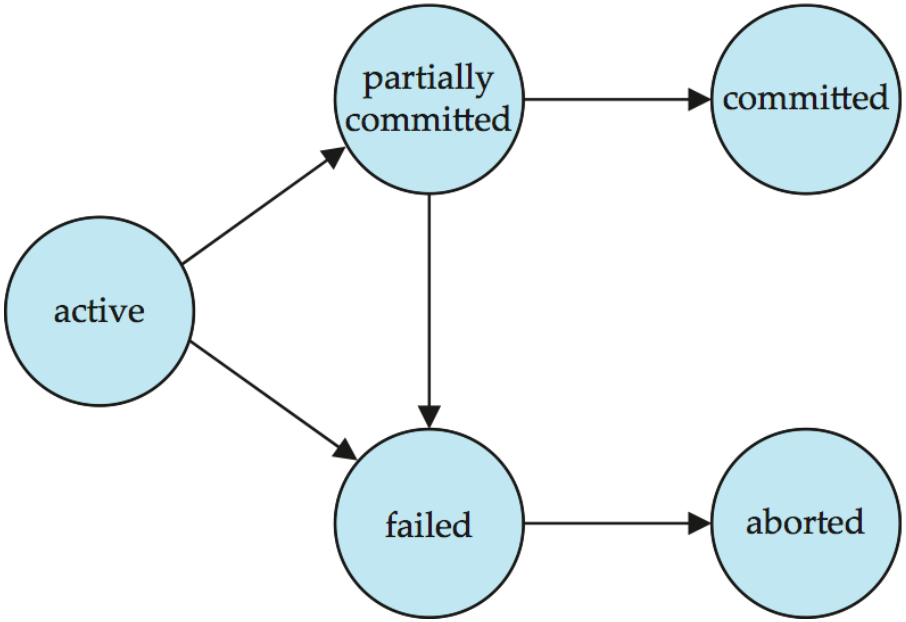

ACID Properties (2)

- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
  - Intermediate transaction results must be hidden from other concurrently executed transactions.
  - Isolation can be ensured trivially by running transactions **serially**
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.


Transaction State (1)

- **Active:** the initial state; the transaction stays in this state while it is executing
- **Partially committed:** after the final statement has been executed
- **Failed:** discover that normal execution can no longer proceed
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
  - kill the transaction
- **Committed** – after successful completion.

Transaction State (2)




Concurrency Control

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - increased processor and disk utilization, leading to better transaction *throughput*
    - e.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - reduced average response time for transactions: short transactions need not wait behind long ones.
- Concurrency control schemes – mechanisms to achieve isolation


Schedules (1)

- Schedule: a sequence of instructions that specify the *chronological* order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of *all* instructions of those transactions
  - must preserve the order in which the instructions appear in *each* individual transaction.

Schedules (2)

- Example of two transactions:
  - T_1 transfers $50 from A to B
  - T_2 transfers 10% of the balance from A to B.
- Serial schedule, T_1 followed by T_2

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedules (3)

- Example of two transactions:
  - T_1 transfers $50 from A to B
  - T_2 transfers 10% of the balance from A to B.
- Serial schedule, T_2 followed by T_1

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

Schedules (4)

- Example of two transactions:
  - T_1 transfers $50 from A to B
  - T_2 transfers 10% of the balance from A to B.
- Concurrent schedule
- Equivalent to previous schedules. Preserves consistency. Why?

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedules (5)

- Example of two transactions:
  - T_1 transfers $50 from *A* to *B*
  - T_2 transfers 10% of the balance from *A* to *B.*
- Concurrent schedule
- Does not preserve consistency. Why?

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit


Serializability (1)

- **Goal:** have concurrent schedules that preserve consistency.
- Assumption: each transaction preserves database consistency.
  - Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is *serializable* if it is *equivalent* to a serial schedule.
- Cannot always determine “equivalency” because we have to understand the semantics of the transactions.
- However, we can allow instructions of different transactions to execute under certain rules, which *guarantee* equivalency.
- Different forms of schedule equivalence give rise to the notions of:
  - conflict serializability
  - view serializability


Conflict Serializability (1)

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  - I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  - I_i = **read**(Q), I_j = **write**(Q). They conflict.
  - I_i = **write**(Q), I_j = **read**(Q). They conflict
  - I_i = **write**(Q), I_j = **write**(Q). They conflict
- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (2)

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)


Concurrency Control, Locking (1)

- A lock is a mechanism to control concurrent access to a data item (record, table, database)
- A lock can be either a “write lock” or “read lock.”
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.
- A *locking protocol* is a *set of rules* followed by all transactions while requesting/releasing locks.
- Locking protocols restrict the set of possible schedules.
- Problems:
  - deadlocks
  - starvation


Concurrency Control, Locking (2)

- Two-phase locking (2PL) – the *standard* in db industry
- 2PL is a protocol which ensures conflict-serializable schedules
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- 2PL assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).


Concurrency Control, MultiVersion (MV) (1)

- Multiversion schemes keep old versions of data item to increase concurrency.
- Each successful write results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a read(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction, and return the value of the selected version
  - reads never have to wait as an appropriate version is returned immediately.

Concurrency Control, MultiVersion (MV) (2)

- Each data item Q has a sequence of versions $<Q_1, Q_2, ..., Q_m>$ Each version Q_k contains three data fields:
  - Content -- the value of version Q_k.
  - W-timestamp(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - R-timestamp(Q_k) -- largest timestamp of a transaction that successfully read version Q_k
- When a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialized to TS(T_i).
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).


Recovery

- Transaction management component implements recovery algorithms.
- Recovery algorithms have two parts:
  - Actions taken during normal transaction processing to ensure enough information exists to recover from failures (logs)
  - Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability (scan logs)


Recovery – Log based (1)

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- Two approaches:
  - log-based recovery
  - shadow-paging
- Focus on log-based recovery
- Assume only sequential transactions – no concurrency

Recovery – Log based (2)

- A log is kept on stable storage.
  - The log is a sequence of log records, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing:
  - <T_i start> log record
- *Before* T_i executes write(X), a log record:
  - <T_i, X, V_1, V_2> is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X. The meaning is that T_i has performed a write on data item X; X had value V_1 before the write, and will have value V_2 after the write.
- When T_i finishes it last statement, the log record:
  - <T_i commit> is written.


Recovery – Log based (3)

- Two approaches using logs
  - Deferred database modification
  - Immediate database modification
- Focus on deferred database modification
- The deferred database modification scheme records all modifications to the log, but defers all the writes to after partial commit.


Recovery – Log based (4)

- Transaction starts by writing <T_i *start*> record to log
- A write(X) operation results in a log record <T_i, X, V> being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- The write is not performed on X at this time (no database update), but is deferred
- When T_i partially commits, <T_i commit> is written to the log.
- Finally, the log records are read and used to actually execute the previously deferred writes


Recovery – Log based (5)

- During recovery after a crash, a transaction needs to be redone if and only if both <T_i start> and<T_i commit> are there in the log
- Redoing a transaction T_i (redo T_i) sets the value of all data items updated by the transaction to the new values
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken

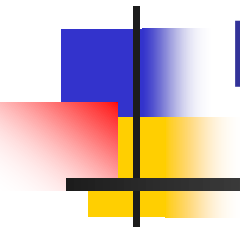
Recovery – Log based (6)

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

- If log on stable storage at time of crash is as in case:
  - (a) no redo actions need to be taken
  - (b) redo(T_0) must be performed - <T_0 commit> is present
  - (c) redo(T_0) must be performed, followed by redo(T_1)


Transactions and Overhead

- Recovery algorithms maintain a log to ensure consistency and atomicity.
- Transactions have to acquire locks to access data items and follow a protocol.
- Transaction management incurs a great *overhead* to the database system.
- Various levels of consistency.
- Think: What about distributed databases? Replication?



Parallel Databases

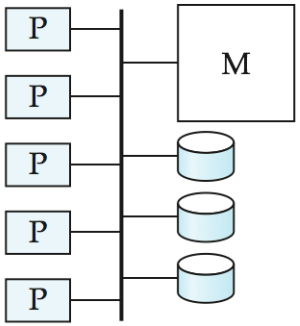

Parallel Database Systems

- Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network.
- A massively parallel or fine grain parallel machine utilizes thousands of smaller processors.
- Two main performance measures:
  - throughput --- the number of tasks that can be completed in a given time interval
  - response time --- the amount of time it takes to complete a single task from the time it is submitted


Parallel Database Architectures (1)

- **Shared memory** -- processors share a common memory
- **Shared disk** -- processors share a common disk
- **Shared nothing** -- processors share neither a common memory nor common disk
- **Hierarchical** -- hybrid of the above architectures

Parallel Database Architectures (2)



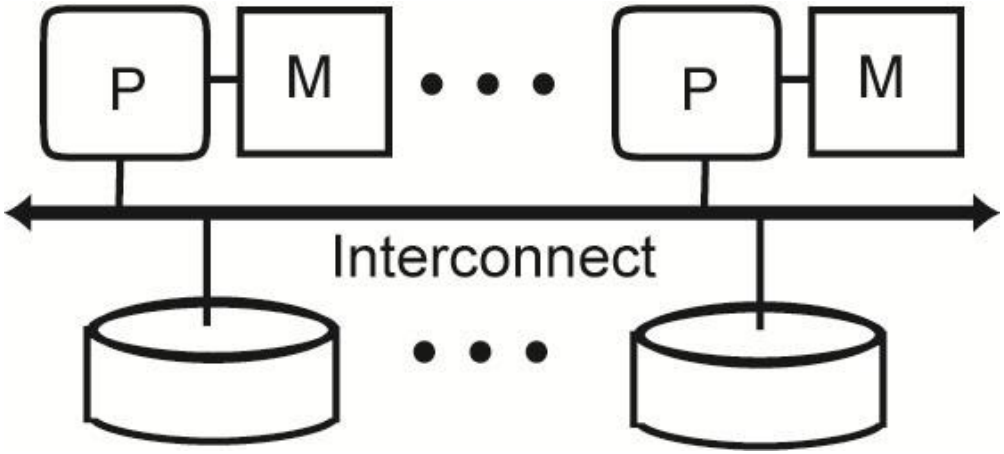
(a) shared memory


Shared Memory

- Processors and disks have access to a common memory, typically via a bus or through an interconnection network
- Extremely efficient communication between processors — data in shared memory can be accessed by any processor without having to move it using software
- Downside – architecture is not scalable beyond 32 or 64 processors since the bus or the interconnection network becomes a bottleneck
- Widely used for lower degrees of parallelism (4 to 8)

Shared Disk (1)

- All processors can directly access all disks via an interconnection network, but the processors have private memories

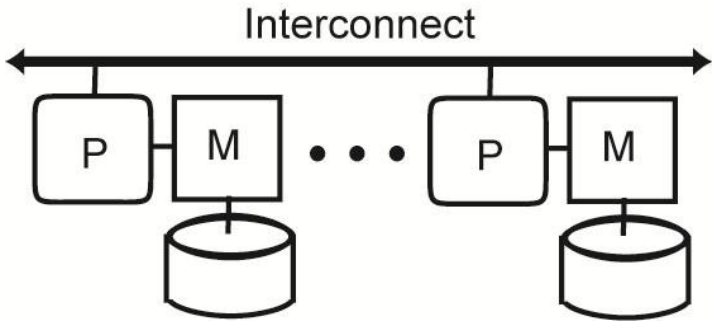



Shared Disk (2)

- The memory bus is not a bottleneck
- Architecture provides a degree of fault-tolerance — if a processor fails, the other processors can take over its tasks since the database is resident on disks that are accessible from all processors
- Bottleneck now at interconnection to the disk subsystem
- Shared-disk systems can scale to a somewhat larger number of processors, but communication between processors is slower

Shared Nothing (1)

- Node consists of a processor, memory, and one or more disks. Processors at one node communicate with another processor at another node using an interconnection network. A node functions as the server for the data on the disk or disks the node owns.
- Examples: Teradata, Tandem, Oracle-n CUBE

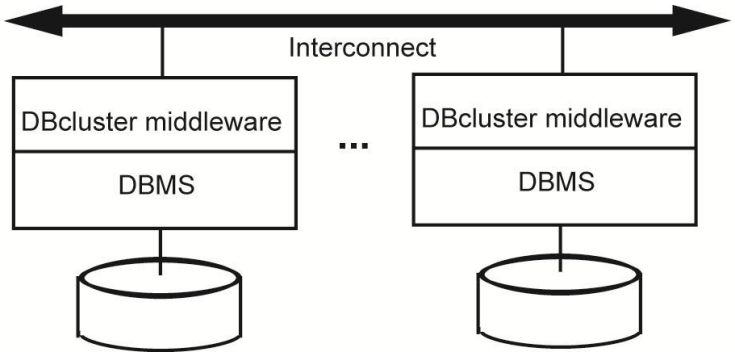



Shared Nothing (2)

- Data accessed from local disks (and local memory accesses) do not pass through interconnection network, thereby minimizing the interference of resource sharing.
- Shared-nothing multiprocessors can be scaled up to thousands of processors without interference.
- Main drawback: cost of communication and non-local disk access; sending data involves software interaction at both ends.

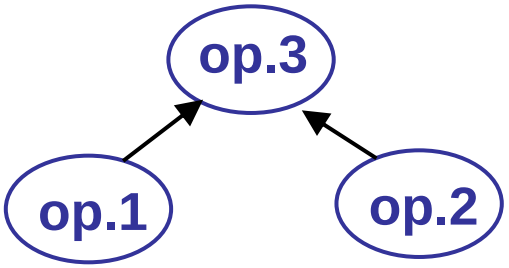
Cluster

- Combines good load balancing of SM with extensibility of SN
- Server nodes: off-the-shelf components
  - From simple PC components to more powerful SMP
  - Yields the best cost/performance ratio
  - In its cheapest form,
- Fast standard interconnect (e.g., Myrinet and Infiniband) with high bandwidth (Gigabits/sec) and low latency

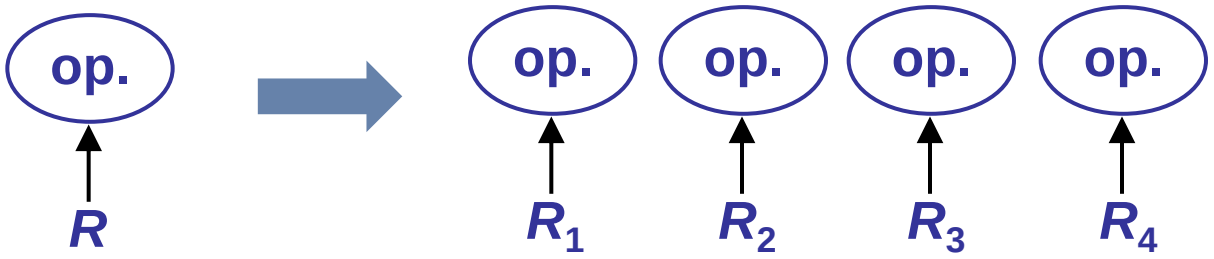


Data-based Parallelism

- Inter-operation
  - p operations of the same query in parallel

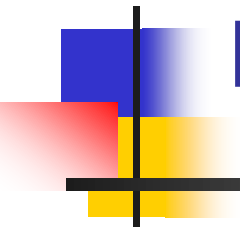


- Intra-operation
  - The same op in parallel




Parallel DBMS Techniques

- Data placement
  - Physical placement of the DB onto multiple nodes
  - Static vs. Dynamic
- Parallel data processing
  - Select is easy
  - Join (and all other non-select operations) is more difficult
- Parallel query optimization
  - Choice of the best parallel execution plans
  - Automatic parallelization of the queries and load balancing
- Transaction management
  - Similar to distributed transaction management



Distributed Databases


Distributed Database Systems

- A distributed database system consists of loosely coupled sites that share no physical component
- Database systems that run on each site are independent of each other
- Transactions may access data at one or more sites


Dimensions of the Problem

- Distribution
  - Whether the components of the system are located on the same machine or not
- Heterogeneity
  - Various levels (hardware, communications, operating system)
  - DBMS important one
    - data model, query language, transaction management algorithms
- Autonomy
  - Not well understood and most troublesome
  - Various versions
    - Design autonomy: Ability of a component DBMS to decide on issues related to its own design.
    - Communication autonomy: Ability of a component DBMS to decide whether and how to communicate with other DBMSs.
    - Execution autonomy: Ability of a component DBMS to execute local operations in any manner it wants to.


Homogeneous Distributed Databases

- In a homogeneous distributed database
  - All sites have identical software
  - Are aware of each other and agree to cooperate in processing user requests.
  - Each site surrenders part of its autonomy in terms of right to change schemas or software
  - Appears to user as a single system
- In a heterogeneous distributed database
  - Different sites may use different schemas and software
    - Difference in schema is a major problem for query processing
    - Difference in software is a major problem for transaction processing
  - Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing


Distributed Data Storage

- Assume relational data model
- Replication
  - System maintains multiple copies of data, stored in different sites, for **faster retrieval** and **fault tolerance**.
- Fragmentation
  - Relation is partitioned into several fragments stored in distinct sites
- Replication and fragmentation can be combined
  - Relation is partitioned into several fragments: system maintains several identical replicas of each such fragment.


Data Replication (1)

- A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites.
- **Full replication** of a relation is the case where the relation is stored at all sites.
- **Fully redundant databases** are those in which every site contains a copy of the entire database.


Data Replication (2)

- Advantages of Replication
  - **Availability**: failure of site containing relation *r* does not result in unavailability of *r* if replicas exist.
  - **Parallelism**: queries on *r* may be processed by several nodes in parallel.
  - **Reduced data transfer**: relation *r* is available locally at each site containing a replica of *r*.


Data Replication (3)

- Disadvantages of Replication
  - Increased cost of updates: each replica of relation *r* must be updated.
  - Increased complexity of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.
    - One solution: choose one copy as **primary copy** and apply concurrency control operations on primary copy


Data Fragmentation

- Division of relation r into fragments r_1, r_2, …, r_n which contain sufficient information to reconstruct relation r.
- **Horizontal fragmentation**: each tuple of *r* is assigned to one or more fragments
- **Vertical fragmentation**: the schema for relation *r* is split into several smaller schemas
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property.
  - A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key.

Horizontal Fragmentation of *account* Relation

branch_name	*account_number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$$account_1 = \sigma_{branch\_name="Hillside"}(account)$$

branch_name	*account_number*	*balance*
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

$$account_2 = \sigma_{branch\_name="Valleyview"}(account)$$

Vertical Fragmentation of *employee_info* Relation

branch_name	*customer_name*	*tuple_id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$deposit_1 = \Pi_{branch\_name,\ customer\_name,\ tuple\_id}\ (employee\_info)$

account_number	*balance*	*tuple_id*
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

$deposit_2 = \Pi_{account\_number,\ balance,\ tuple\_id}\ (employee\_info)$


Advantages of Fragmentation

- Horizontal:
  - allows parallel processing on fragments of a relation
  - allows a relation to be split so that tuples are located where they are most frequently accessed
- Vertical:
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation
- Vertical and horizontal fragmentation can be mixed.
  - Fragments may be successively fragmented to an arbitrary depth


Data Transparency

- Data transparency: Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system
- Consider transparency issues in relation to:
  - Fragmentation transparency
  - Replication transparency
  - Location transparency


Naming of Data Items - Criteria

1. Every data item must have a system-wide unique name
2. It should be possible to find the location of data items efficiently
3. It should be possible to change the location of data items transparently
4. Each site should be able to create new data items autonomously

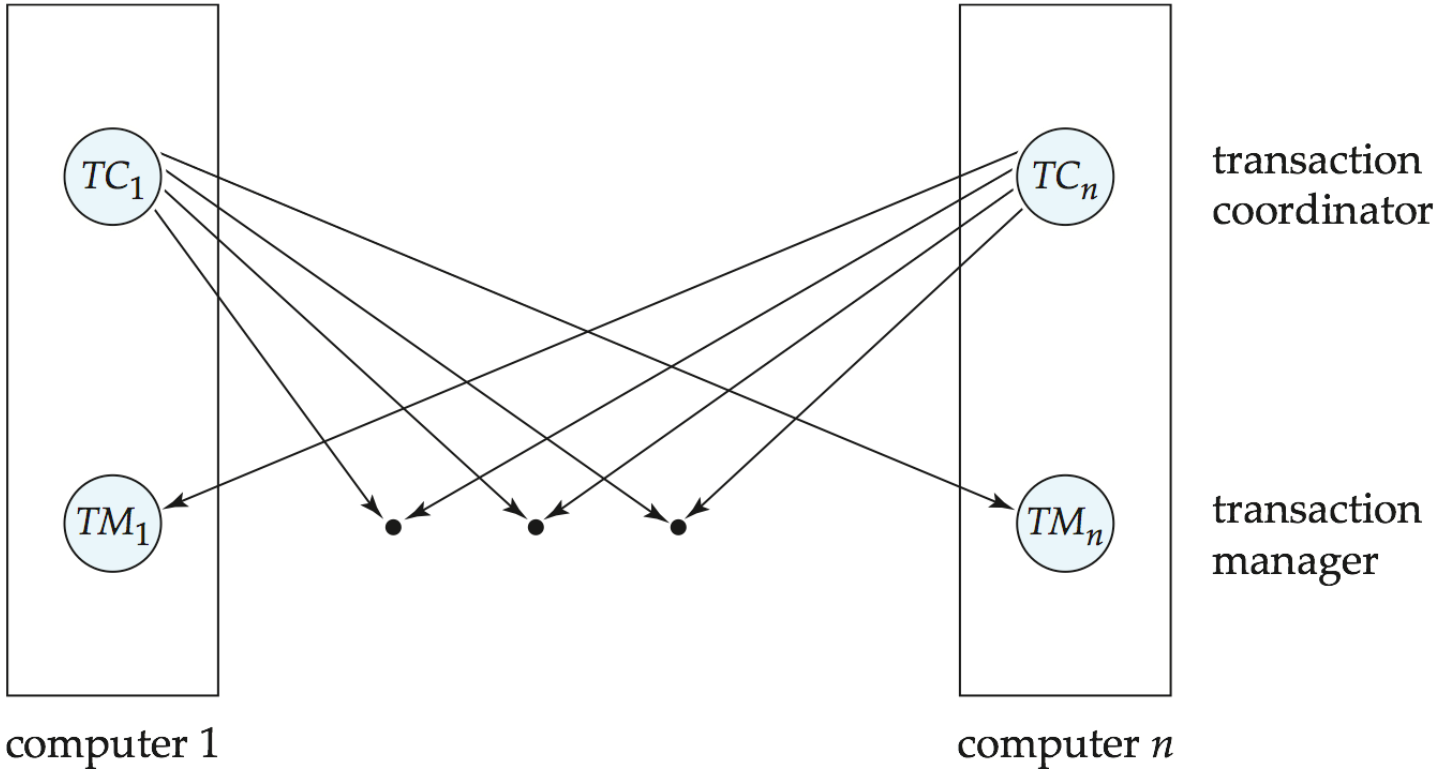

Centralized Scheme – Name Server

- Structure:
  - the name server assigns all names
  - each site maintains a record of local data items
  - sites ask name server to locate non-local data items
- Advantages:
  - satisfies naming criteria 1-3
- Disadvantages:
  - does not satisfy naming criterion 4
  - name server is a potential **performance bottleneck**
  - name server is a **single point of failure**


Distributed Transactions

- Transaction may access data at several sites.
- Each site has a local **transaction manager** responsible for:
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing **at that** site
- Each site has a **transaction coordinator**, which is responsible for:
  - Starting the execution of transactions that **originate** at the site
  - Distributing subtransactions at appropriate sites for execution
  - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites

Distributed Transactions - Architecture




System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site
  - Loss of messages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
  - Network partition
    - A network is said to be partitioned when it has been split into two or more subsystems that lack any connection between them; a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.


Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2PC) protocol is widely used
- The *three-phase commit* (3PC) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol. This protocol is not used in practice.


Two Phase Commit Protocol (2PC)

- Assumes **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i


Phase 1: Obtaining a Decision

- Coordinator asks all participants to ***prepare*** to commit transaction *T*.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
    - add the record <**ready** *T*> to the log
    - force *all records* for *T* to stable storage
    - send **ready** *T* message to C_i

Phase 2: Recording the Decision

- *T* can be committed if C_i received a **ready** *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.


Concurrency Control

- Modify concurrency control schemes for use in distributed environment.
- We assume that each site participates in the execution of a commit protocol to ensure global transaction automicity.
- We assume all replicas of any item are updated
  - Will see how to relax this in case of site failures later


Single-Lock-Manager Approach (1)

- System maintains a *single* lock manager that resides in a *single* chosen site, say S_i
- When a transaction needs to lock a data item, it sends a lock request to S_i and lock manager determines whether the lock can be granted immediately
  - If yes, lock manager sends a message to the site which initiated the request
  - If no, request is delayed until it can be granted, at which time a message is sent to the initiating site

Single-Lock-Manager Approach (2)

- The transaction can read the data item from *any* one of the sites at which a replica of the data item resides.
- Writes must be performed on all replicas of a data item
- Advantages of scheme:
  - Simple implementation
  - Simple deadlock handling
- Disadvantages of scheme are:
  - Bottleneck: lock manager site becomes a bottleneck
  - Vulnerability: system is vulnerable to lock manager site failure


Distributed Lock Manager

- In this approach, functionality of locking is implemented by lock managers at each site
  - Lock managers control access to local data items
    - But special protocols may be used for replicas
- Advantage: work is distributed and can be made robust to failures
- Disadvantage: deadlock detection is more complicated
  - Lock managers cooperate for deadlock detection
    - More on this later
- Several variants of this approach
  - Primary copy
  - Majority protocol
  - Biased protocol
  - Quorum consensus


Replication with Weak Consistency (1)

- Many commercial databases support replication of data with weak degrees of consistency (i.e., without a guarantee of serializabiliy)
- E.g., **master-slave replication**: updates are performed at a single "master" site, and propagated to "slave" sites.
  - Propagation is not part of the update transaction: its is decoupled
    - May be immediately after transaction commits
    - May be periodic
  - Data may only be read at slave sites, not updated
    - No need to obtain locks at any remote site
  - Particularly useful for distributing information
    - E.g., from central office to branch-office
  - Also useful for running read-only queries offline from the main database


Replication with Weak Consistency (2)

- Replicas should see a **transaction-consistent snapshot** of the database
  - That is, a state of the database reflecting all effects of all transactions up to some point in the serialization order, and no effects of any later transactions.
- E.g., Oracle provides a **create snapshot** statement to create a snapshot of a relation or a set of relations at a remote site
  - snapshot refresh either by recomputation or by incremental update
  - Automatic refresh (continuous or periodic) or manual refresh


Availability

- High availability: time for which system is not fully usable should be extremely low (e.g., 99.99% availability)
- Robustness: ability of system to function despite of failures of components
- Failures are more likely in large distributed systems
- To be robust, a distributed system must
  - Detect failures
  - Reconfigure the system so computation may continue
  - Recovery/reintegration when a site or link is repaired
- Failure detection: distinguishing link failure from site failure is hard
  - (partial) solution: have multiple links, multiple link failure is likely a site failure


Reconfiguration (1)

- Reconfiguration:
  - Abort all transactions that were active at a failed site
    - Making them wait could interfere with other transactions since they may hold locks on other sites
    - However, in case only some replicas of a data item failed, it may be possible to continue transactions that had accessed data at a failed site (more on this later)
  - If replicated data items were at failed site, update system catalog to remove them from the list of replicas.
    - This should be reversed when failed site recovers, but additional care needs to be taken to bring values up to date
  - If a failed site was a central server for some subsystem, an **election** must be held to determine the new server
    - E.g., name server, concurrency coordinator, global deadlock detector


Reconfiguration (2)

- Since network partition may not be distinguishable from site failure, the following situations must be avoided
  - Two ore more central servers elected in distinct partitions
  - More than one partition updates a replicated data item
- Updates must be able to continue even if some sites are down
- Solution: majority based approach
  - Alternative of “read one write all available” is tantalizing but causes problems


Distributed Query Processing

- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.
- In a distributed system, other issues must be taken into account:
  - The cost of a data transmission over the network
  - The potential gain in performance from having several sites process parts of the query in parallel

Query Transformation (1)

- Translating algebraic queries on fragments.
  - It must be possible to construct relation *r* from its fragments
  - Replace relation *r* by the expression to construct relation *r* from its fragments
- Consider the horizontal fragmentation of the *account* relation

  $account_1 = \sigma_{branch\_name = \text{“Hillside”}} (account)$

  $account_2 = \sigma_{branch\_name = \text{“Valleyview”}} (account)$
- The query $\sigma_{branch\_name = \text{“Hillside”}} (account)$ becomes

  $\sigma_{branch\_name = \text{“Hillside”}} (account_1 \cup account_2)$

  which is optimized into

  $\sigma_{branch\_name = \text{“Hillside”}} (account_1) \cup \sigma_{branch\_name = \text{“Hillside”}} (account_2)$

Query Transformation (2)

- Since $account_1$ has only tuples pertaining to the Hillside branch, we can eliminate the selection operation.
- Apply the definition of $account_2$ to obtain

  $\sigma_{branch\_name = \text{“Hillside”}} (\sigma_{branch\_name = \text{“Valleyview”}} (account))$
- This expression is the empty set regardless of the contents of the *account* relation.
- Final strategy is for the Hillside site to return $account_1$ as the result of the query.


Simple Join Processing

- Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented

  $account \bowtie depositor \bowtie branch$

- *account* is stored at site S_1
- *depositor* at S_2
- *branch* at S_3
- For a query issued at site S_I, the system needs to produce the result at site S_I


Possible Query Processing Strategies

- Ship copies of all three relations to site S_I and choose a strategy for processing the entire locally at site S_I.
- Ship a copy of the account relation to site S_2 and compute $temp_1 = account \bowtie depositor$ *at* S_2. Ship $temp_1$ from S_2 to S_3, and compute $temp_2 = temp_1 \bowtie branch$ at S_3. Ship the result $temp_2$ to S_I.
- Devise similar strategies, exchanging the roles S_1, S_2, S_3
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site


Join Strategies that Exploit Parallelism

- Consider $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$ where relation r_i is stored at site S_i.

  The result must be presented at site S_1.
- r_1 is shipped to S_2 and $r_1 \bowtie r_2$ is computed at S_2: simultaneously r_3 is shipped to S_4 and $r_3 \bowtie r_4$ is computed at S_4
- S_2 ships tuples of $(r_1 \bowtie r_2)$ to S_1 as they produced;
  S_4 ships tuples of $(r_3 \bowtie r_4)$ to S_1
- Once tuples of $(r_1 \bowtie r_2)$ and $(r_3 \bowtie r_4)$ arrive at S_1 $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$ is computed in parallel with the computation of $(r_1 \bowtie r_2)$ at S_2 and the computation of $(r_3 \bowtie r_4)$ at S_4.


Heterogeneous Distributed Databases

- Many database applications require data from a variety of preexisting databases located in a heterogeneous collection of hardware and software platforms
- Data models may differ (hierarchical, relational, etc.)
- Transaction commit protocols may be incompatible
- Concurrency control may be based on different techniques (locking, timestamping, etc.)
- System-level details almost certainly are totally incompatible.
- A **multidatabase system** is a software layer on top of existing database systems, which is designed to manipulate information in heterogeneous databases
  - Creates an illusion of logical database integration without any physical database integration


Mediator Systems

- **Mediator** systems are systems that integrate multiple heterogeneous data sources by providing an integrated global view, and providing query facilities on global view
  - Unlike full fledged multidatabase systems, mediators generally do not bother about transaction processing
  - But the terms mediator and multidatabase are sometimes used interchangeably
  - The term **virtual database** is also used to refer to mediator/multidatabase systems


Advantages

- Preservation of investment in existing
  - hardware
  - system software
  - applications
- Local autonomy and administrative control
- Allows use of special-purpose DBMSs
- Step towards a unified homogeneous DBMS
  - Full integration into a homogeneous DBMS faces
    - technical difficulties and cost of conversion
    - organizational/political difficulties
      - organizations do not want to give up control on their data
      - local databases wish to retain a great deal of autonomy


Unified View of Data

- Agreement on a common data model
  - Typically the relational model
- Agreement on a common conceptual schema
  - Different names for same relation/attribute
  - Same relation/attribute name means different things
- Agreement on a single representation of shared data
  - E.g., data types, precision,
  - Character sets (ASCII vs EBCDIC, sort order variations)
- Agreement on units of measure
- Variations in names
  - E.g., Köln vs Cologne, Mumbai vs Bombay


Query Processing (1)

- Several issues in query processing in a heterogeneous database
- Schema translation
  - Write a **wrapper** for each data source to translate data to a global schema
  - Wrappers must also translate updates on global schema to updates on local schema


Query Processing (2)

- Limited query capabilities
  - Some data sources allow only restricted forms of selections
    - E.g., web forms, flat file data sources
  - Queries have to be broken up and processed partly at the source and partly at a different site
- Removal of duplicate information when sites have overlapping information
  - Decide which sites to execute query
- Global query optimization


Big Data Management & Multidatabases

- Variety in data formats
  - structured (relational), semi-structured, textual, unstructured
- Variety in data management systems
  - RDBMS, HDFS, NoSQL, Stream engines
- Variety in data models
  - relational, semi-structured, OO, graph, hashmaps, etc.
- How do we “integrate” all these systems to form “one” view? (later in the course…)