Advanced Database Management Systems

Database Management Systems

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Database Management Systems

What are they?

Definition

A **database management system (DBMS)** is

- a software package
- for supporting applications
- aimed at managing very large volumes of data
- efficiently and reliably
- transparently with respect to the underlying hardware and network infrastructures
- and projecting a coherent, abstract model
- of the underlying reality of concern to applications
- through high-level linguistic abstractions.

Managing Very Large Volumes of Data

What does it mean?

managing storing for querying (often, updating as well, and more recently searching, exploring, discovering)

very large for a given underlying hardware and network infrastructure, volumes that require special strategies to enable scale-out in storage with no scale-down in processing are **very large**

data basically, **facts** describing an **entity** (e.g., the employee with id='123' has name='Jane', the employee with id='456' has name='John'), grouped in **collections** (e.g., Employee) and **related** to one another (e.g., Jane manages John)

Projecting a Coherent Abstract Model

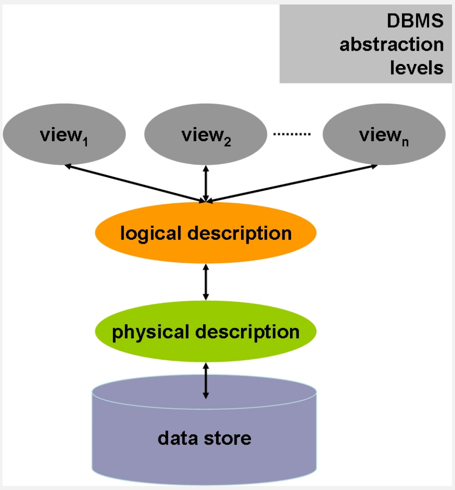
What does it mean?

- A **data model** comprises a set of abstract, domain-independent concepts with which data in the database can be described.
- Such concepts (e.g., primary/foreign keys) induce **integrity constraints** (e.g., referential ones, i.e., a foreign key in one relation must appear as primary key in another).
- A **schema** describes the domain-specific concepts that go into a database in terms of the domain-independent concepts available in the given data model.
- A database **instance** (i.e., the database state at a point in the life cycle of the database) is a snapshot of the world as captured in data and must always be valid with respect to the schema.
- Each DBMS supports one data model (e.g., the relational model), but many supported data models subsume others (e.g., the object-relational model).

High-Level

What does it mean?

- **physical schema**: storage-level structures
- **conceptual/logical schema**: domain-specific, application-independent concepts organized as a collection of **entities** (e.g., relations) and **relationships** (e.g., expressed by means of primary/foreign keys)
- **external schema**: application-specific concepts/requirements as a collection of **views/queries** over the logical schema



Linguistic Abstractions (1)

What does it mean?

A DBMS typically supports three sub-languages:

- DDL A **data definition language** to formulate schema-level concepts.
- DML A **data manipulation language** to formulate changes to be effected in a database instance.
- (D)QL A **(data) query language** to formulate retrieval requests over a database instance.

Linguistic Abstractions (2)

What does it mean?

The best known DBMS language is SQL, different parts of which serve as DDL, DML and QL for (primarily) relational DBMSs.

- The goal of a DDL is to describe application-independent notions (i.e., at the physical and logical levels).
- The goal of a DML and a (D)QL is describe application-specific notions (i.e., at the view level).

```
CREATE TABLE
  Employee
  (id CHAR(3),name VARCHAR(30),branch VARCHAR(10))
```

```
INSERT INTO
   Employee
   VALUES ('123','Jane','Manchester')
INSERT INTO
   Employee
   VALUES ('456','John','Edinburgh')
```

```
SELECT id
FROM   Employee
WHERE  branch = 'Manchester'
```

```
CREATE VIEW
   ManchesterEmployees
AS
   SELECT id
   FROM   Employee
   WHERE  branch = 'Manchester'
```

Database Languages (1)
Are they any different?

- Database languages are different, by design, from general-purpose programming languages.
- Each sub-language caters for distinct needs that in a general-purpose programming language are not factored out.
  - DDL statements update the stored metadata schema-level concepts and correspond to **variable and function declarations** (but have side-effects).
  - DML statements update the database, i.e., change (equivalently, cause a transition from) one database instance into another, and correspond to **assignments**.
  - DQL expressions do not have side-effects, and correspond to **(pure) expressions**.
- Database languages are designed to operate on collections, without explicit iteration, whereas most general-purpose programming language are founded on explicit iteration.

Database Languages (2)
Are they any limited?

- QLs are limited, by design, compared with general-purpose programming languages.
- They are not Turing-complete.
- Even (ANSI) SQL was not Turing-complete until procedural constructs (i.e., so called persistent stored modules) were introduced to make it so.
- They are not meant for complex calculations nor for operating on anything other than large collections.
- These limitations make QLs declarative and give them simple formal semantics (in calculus and in algebraic form).
- These properties in turn make it possible for a declarative query to be rewritten by an optimizer into an efficient procedural execution plan.

Summary
Database Management Systems

- The added-value that DBMSs deliver to organizations stems from controlled use of abstraction.
- The adoption of application-independent data models provides a common formal framework upon which several levels of description become available.
- Such descriptions are conveyed in formal languages that separate concerns and facilitate the implementation of efficient evaluation engines.

Advanced Database Management Systems
Architecture: Classical Case and Variations

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

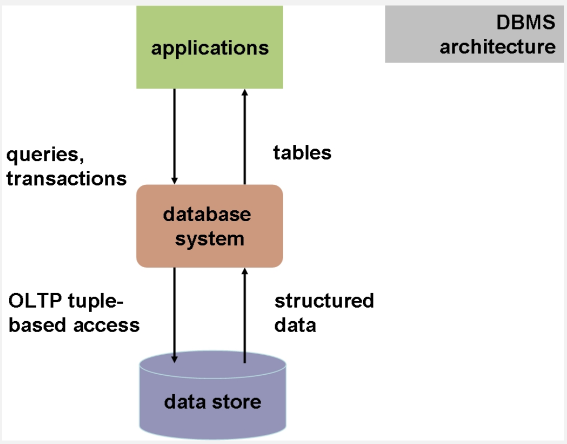
The Classical Case

Strengths and Weaknesses of Classical DBMSs

Variations on the Classical Case

The Architecture of DBMS-Centred Applications
Three Tiers

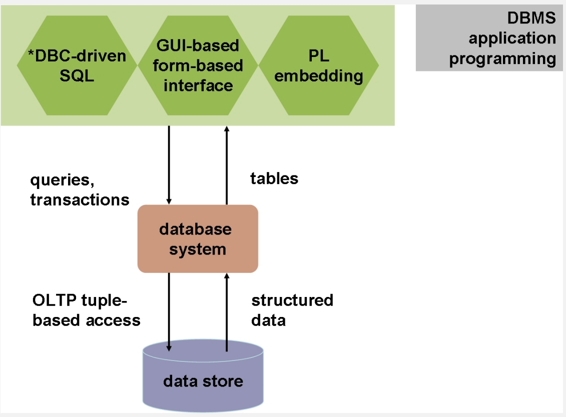
- A classical DBMS supports **on-line transaction processing (OLTP)** applications.
- Applications send queries and transactions that the DBMS converts into OLTP tuple-based operations over the data store.
- What comes back is structured data (e.g., records) as tables, i.e., collections of tuples.



Application Interfaces

Three Routes

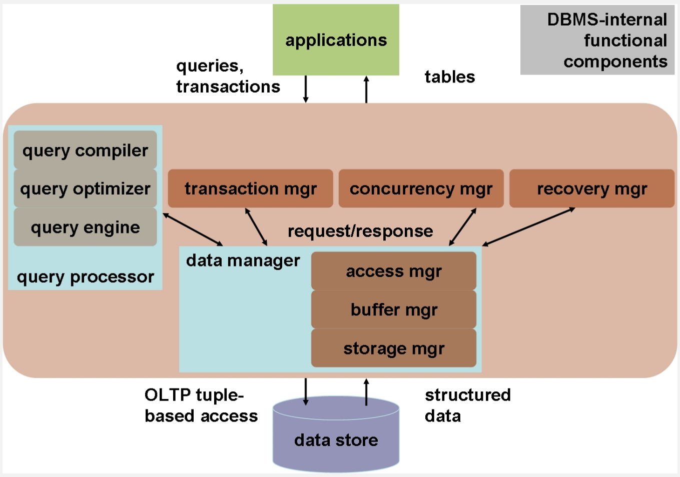
- Occasional, non-knowledgeable users are served by graphical user interfaces (GUI), which are often form-based.
- Most application programming tends to benefit from database connectivity middleware (e.g., ODBC, JDBC, JDO, etc.).
- If necessary, from a general-purpose programming language, applications can invoke DBMS services.



Internal Architecture of Classical DBMSs

Four+One Main Service Types

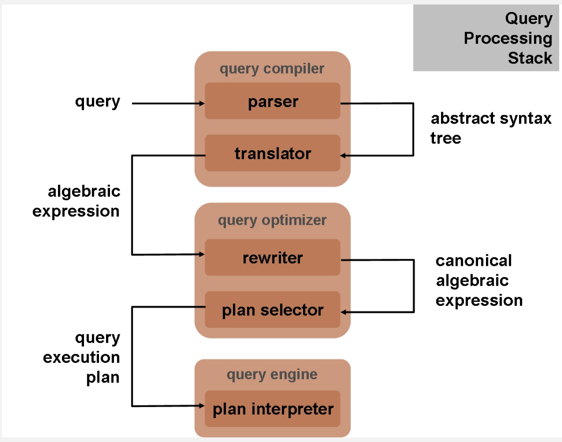
- Query Processing
- Transaction Processing
- Concurrency Control
- Recovery
- Storage Management



The Query Processing Stack

Declarative-to-Procedural, Equivalence-Preserving Program Transformation

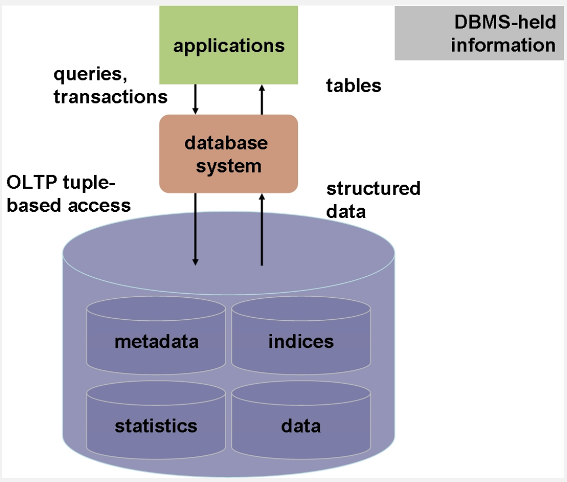
1. **Parse** the declarative query
2. **Translate** to obtain an algebraic expression
3. **Rewrite** into a canonical, heuristically-efficient logical query execution plan (QEP)
4. **Select** the algorithms and access methods to obtain a quasi-optimal, costed concrete QEP
5. **Execute** (the typically interpretable form of) the procedural QEP



Data Store Contents

Four Main Types of Data

- **Metadata** includes schema-level information and a description of the underlying computing infrastructure.
- **Statistics** are mostly information about the trend/summary characteristics of past database instances.
- **Indices** are built for efficient access to the data.
- **Data** is what the database instance contains.



Summary
Classical DBMSs

- Classical DBMSs have been incredibly successful in underpinning the day-to-day life of organizations.
- Their internal functional architecture had not, until very recently, changed much over the last three decades.
- More recently, this architecture has been perceived as being unable to deliver certain kinds of services to business.
- Under pressure from business interests and reacting to opportunities created by new computing infrastructures, classical DBMSs have been transforming themselves into different kinds of advanced DBMSs that this course will explore.

Classical DBMSs: Strengths
What are classical database management systems good at?

- Classical database management systems (DBMSs) have been very successful.
- In the last four decades, they have become an indispensable infrastructural component of organizations.
- They play a key role in reliably and efficiently reflecting the transaction-level unfolding of operations in the value-adding chain of an organization.
- Each transaction (e.g., an airline reservation, a credit card payment, an item sold in a checkout) is processed soundly, reliably, and efficiently.
- Effects are propagated throughout the organization.

Classical DBMSs: Weaknesses
Where do classical database management systems come short?

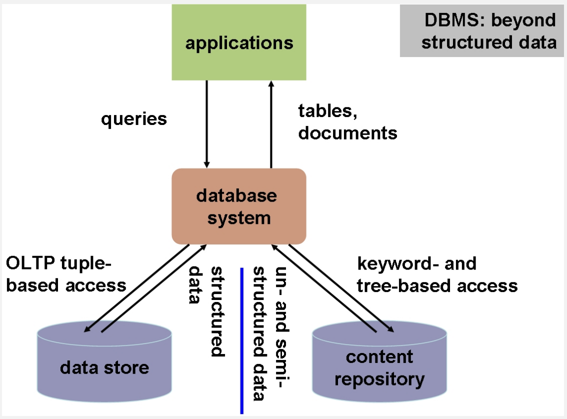
- Classical database management systems (DBMSs) assume that:
  1. Data is structured in the form of records
  2. Only on-line transaction processing (OLTP) is needed
  3. Data and computational resources are centralized
  4. There is central control over central resources
  5. There is no need for dynamically responding in real-time to external events
  6. There is no need for embedding in the physical world in which organizations exist
- This is too constraining for most modern businesses.
- Classical DBMSs support fewer needs of organizations than they used to.

Classical DBMSs: Trends
How are DBMSs evolving?

- Most cutting-edge research in databases is geared towards supporting:
  1. Un- and semi-structured data too
  2. On-line analytical processing (OLAP) too
  3. Distributed data and computational resources
  4. Absence of central control over distributed resources
  5. Dynamic response in real-time to external events
  6. Embedding in the physical world in which the organization exists
- DBMSs that exhibit these capabilities are **advanced** in the sense used here.

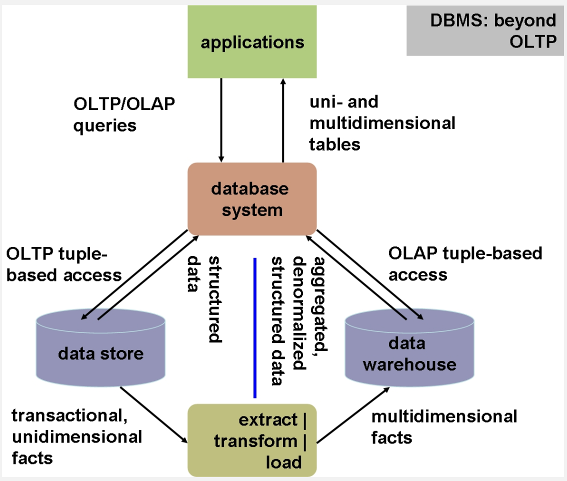
Beyond Structured Data

- Add support to un- and semi-structured data in document form
- Stored in content repositories
- Using keyword-based search and access methods for graph/tree fragments



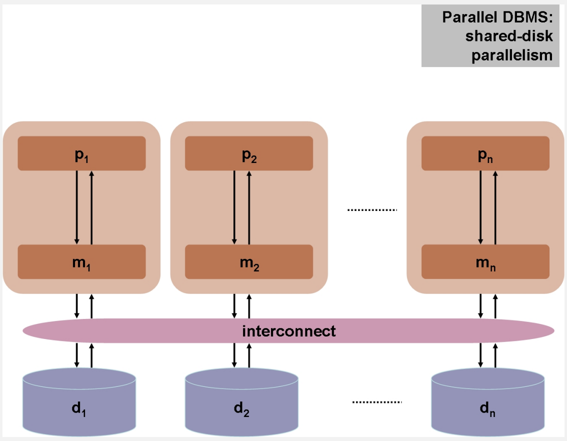
Beyond OLTP

- Preprocess, aggregate and materialize separately
- Add support for OLAP
- Using multidimensional, denormalized logical schemas



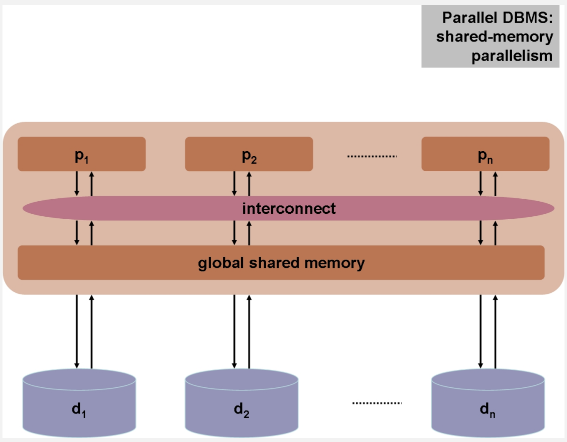
Parallelization (1)
Shared-Disk Parallelism

- Place a fast interconnect between memory and comparatively slow disks
- Parallelize disk usage to avoid secondary-memory contention



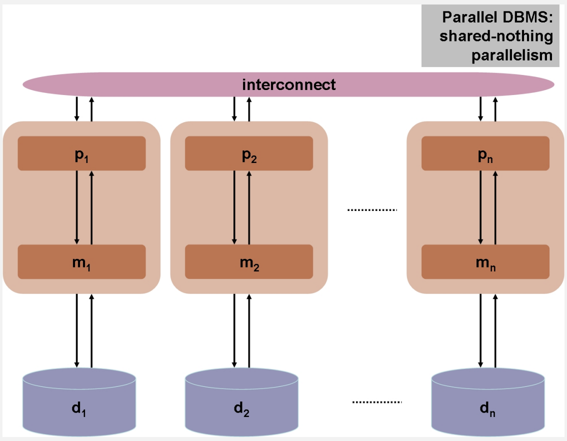
Parallelization (2)
Shared-Memory Parallelism

- Place a fast interconnect between processor and memory
- Parallelize memory usage to avoid primary-memory contention



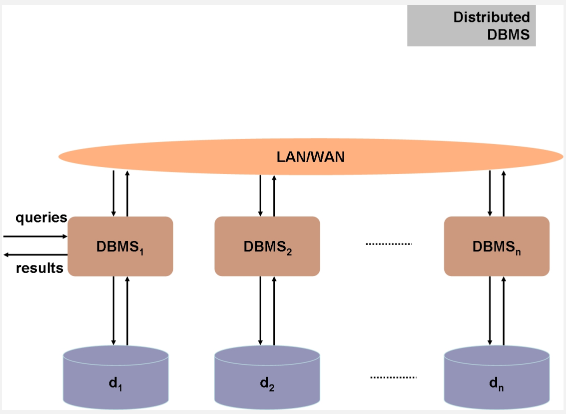
Parallelization (3)
Shared-Nothing Parallelism

- Place a fast interconnect between full processing units
- Parallelize processing using black-boxes that are locally resource-rich



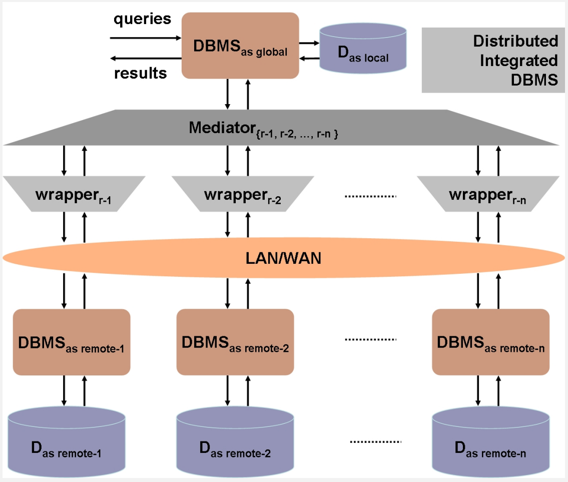
Distribution (1)
Multiple DBMSs

- Harness distributed resources
- Using a full-fledged local or wide-area network as interconnect



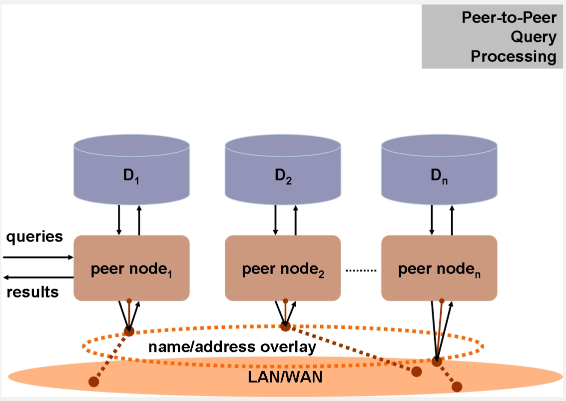
Distribution (2)
Global, Integrated DBMSs

- Renounce central control
- Harness heterogeneous, autonomous, distributed resources
- Project a global view by mediation over wrapper-homogenized local sources



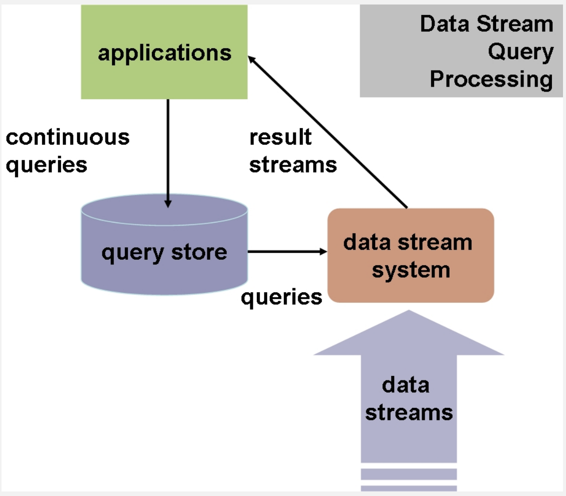
Distribution (3)
Peer-to-Peer DBMSs

- Renounce global view and query expressiveness
- Benefit from inherent scalability over large-scale, extremely-wide-area networks



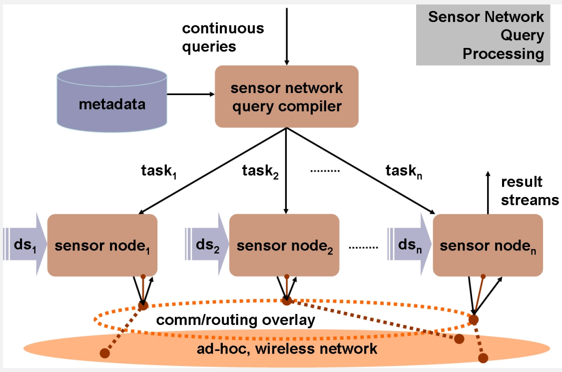
Distribution (4)
Data Stream Management Systems

- Enable dynamic response in real-time to external events
- Placing queries that execute periodically or reactively
- Over data that is pushed onto the system in the form of unbounded streams



Distribution (5)
Sensor Network Data Management

- Embed data-driven processes in the physical world
- Overlaying query processing over an ad-hoc wireless network of intelligent sensor nodes
- Over pull-based data streams



Advanced DBMSs (1)

Why do they matter?

OLAP/DM Companies need to make more, and more complex, decisions more often and more effectively to remain competitive.

Text-/XML-DBMSs The ubiquity and transparency of networks means data can take many forms, is everywhere, and can be processed anywhere.

Advanced DBMSs (2)

Why do they matter?

P/P2P/DDBMSs For both data and computation, provision of resources is now largely servicized and can be negotiated, or harvested.

Stream DMSs Widespread cross-enterprise integration means that companies must be able to respond in real-time to events streaming in from their commercial and financial environment

Sensor DMSs Most companies are aiming to sense and respond not just to the commercial and financial environment but to the physical environment too.

Summary
Advanced DBMSs

- Architecturally, advanced DBMSs characterize different responses to
  - modern functional and non-functional application requirements
  - the availability of advanced computing and networking infrastructures
- Advanced DBMSs are motivated by real needs of modern organizations, in both the industrial and the scientific arena.

Advanced Database Management Systems
The Relational Case: Data Model, Databases, Languages

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

The Relational Model of Data
Why?

- Conceptually simple:
  - one single, collection-valued, domain-independent type
- Formally elegant:
  - a very constrained system of first-order logic with both a model- and a proof-theoretic view
  - gives rise to (formally equivalent) declarative and procedural languages, i.e., the domain and the tuple relational calculi, and the relational algebra, resp.
- Practical:
  - underlies SQL
  - possible to implement efficiently
  - has been so implemented many times
- Flexible:
  - often accommodates useful extensions

Relational Databases (1)

Definitions (1)

Definition

A **relational database** is a set of relations.

Example

D = { Students, Enrolled, Courses, ... }

Definition

A **relation** is defined by its schema and consists of a collection of tuples/rows/records that conform to that schema.

Relational Databases (2)

Definitions (2)

Definition

A **schema** defines the **relation name** and the name and **domain**/type of its **attributes**/columns/fields.

Example

Students (stdid: integer, name: string, login: string, age: integer, gpa: real)

Relational Databases (3)
Definitions (3)

Definition

Given its schema, a **relation (instance)** is a subset of the Cartesian product induced by the domain of its attributes.

Definition

A **tuple** in a relation instance is an element in the Cartesian product defined by the relation schema that the instance conforms to.

Example

stdid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

Relational Databases (4)
Underlying Assumptions

- Relations are classically considered to be a set (hence, all tuples are unique and their order does not determine identity).
- In practice (e.g., in SQL), relations are multisets/bags, i.e., they may contain duplicate tuples, but their order still does not determine identity.

Relational Databases (5)

Definitions (4)

Definition

The number of attributes in a relation schema defines its **arity**/degree.

Definition

The number of tuples in a relation defines its **cardinality**.

Example

- arity(Students) $= 5$
- cardinality(Students) $=$ |Students| $= 3$

Relational Databases (6)

Integrity Constraints (1)

Definition

An **integrity constraint (IC)** is a property that must be true for all database instances.

Example

Domain Constraint: The value of an attribute belongs to the schema-specified domain.

- ICs are specified when the schema is defined.
- ICs are checked when a relation is modified.
- A legal instance of a relation is one that satisfies all specified ICs.
- A DBMS does not normally allow an illegal instance to be stored (or to result from an update operation).

Relational Databases (7)

Integrity Constraints (2)

Definition

1. A set of fields is a **key** for a relation if both:
    1.1 No two distinct tuples can have the same values for those fields.
    1.2 This is not true for any subset of those fields.
2. A **superkey** is not a key, it is a set of fields that properly contains a key.
3. If there is more than one key for a relation, each such key is called a **candidate key**.
4. The **primary key** is uniquely chosen from the candidate keys.

Relational Databases (8)

Integrity Constraints (3)

Example

- {stdid} is a key in Students, so is {login}, {name} is not.
- {stdid, name} is a superkey in Students.
- {stdid} and {login} are candidate keys in Students
- {stdid} may have been chosen to be the primary key out of the candidate keys.

Relational Databases (9)

Integrity Constraints (4)

- A **foreign key** is set of fields in one relation that
  - appears as the primary key in another relation
  - can be used to refer to tuples in that other relation
  - by acting like a logical pointer
  - it expresses a relationship between two entities
- A DBMS does not normally allow an operation whose outcome violates referential integrity.

Relational Databases (10)

Integrity Constraints (5)

Example

Enrolled (cid: string, grade: string, studentid: string)

Example

cid	grade	studentid
CS101	A	53666
MA102	B	53688
BM222	B	53650

- E.g. {studentid} is a foreign key in Enrolled using the {stdid} primary key of Students to refer to the latter.

Relational Databases (11)

Where Do Integrity Constraints Come From?

- ICs are an aspect of how an organization chooses to model its data requirements in the form of database relations.
- Just like a schema must be asserted, so must ICs.
- We can check a database instance to see if an IC is violated, but we cannot infer that an IC is true by looking at an instance.
- An IC is a statement about all possible instances, not about the particular instance we happen to be looking at.

Summary

Relational Databases

- Since their conception in the late 1960s, and particularly after the first successful, industrial-strength implementations appeared in the 1970s, relational databases have attracted a great deal of praise for their useful elegance.
- Over the 1980s, relational databases became the dominant paradigm, a position they still hold (with some notable evolutionary additions).

Relational Query Languages (1)

Why do they matter?

- They were a novel contribution and are a major strength of the relational model.
- They support simple, powerful, well-founded querying of data.
- Requests are specified declaratively and delegated to the DBMS for efficient evaluation.

Relational Query Languages (2)

Why are they special?

- The success of this approach depends on
  - the definition of a pair of query languages, one declarative and one procedural
  - being given a formal semantics that
    - allows their equivalence to be proved
    - the mapping from one to other to be formalized
  - with closure properties (i.e., any expression evaluates to an output of the same type as its arguments) for recursive composition.
- In view of such results, the DBMS can implement a domain-independent query processing stack, such as we have seen in a previous lecture.

Relational Query Languages (3)
Declarative and Procedural, Abstract and Concrete

- By **declarative** we mean a language in which we describe the desired answer without describing how to compute it.
- By **procedural** we mean a language in which we describe the desired answer by describing how to compute it.
- By **abstract** we mean that the language does not have a concrete (let alone, standardized) syntax (and thus, no reference implementation).
- By **concrete** we mean that the language does have a concrete (ideally, standardized) syntax (and thus, often a reference implementation as well).

Relational Query Languages (4)
Domain/Tuple Relational Calculi, Relational Algebra, SQL

- Classically, the relational model defines three expressively-equivalent abstract languages:
  - the **domain relational calculus (DRC)**
  - the **tuple relational calculus (TRC)**
  - the **relational algebra (RA)**
- RA is procedural (more on it later), DRC and TRC are declarative (see the Bibliography for more on those).
- **SQL** (for **Structured Query Language**) is a concrete language whose core is closely related to TRC.

Relational Query Languages (5)

A First Glimpse

Example

Retrieve the gpa of students with age greater than 18.

TRC: $\{A \mid \exists S \in Students\,(S.age > 18 \wedge A.gpa = S.gpa)\}$

RA: $\pi_{gpa}(\sigma_{age>18}(Students))$

SQL: `SELECT S.gpa FROM Students S WHERE S.age > 18`

Answer:

gpa
3.8

Relational Query Languages (6)

What do these queries compute?

Example

TRC: $\{A \mid \exists S \in Students\,\exists E \in Enrolled\,(S.stdid = E.studentid \wedge E.grade = 'B' \wedge A.name = S.name \wedge A.cid = E.cid)\}$

RA: $\pi_{name,cid}(\sigma_{grade='B'}(Students \bowtie_{stdid=studentid} Enrolled))$

SQL:
```
SELECT S.name, E.cid
FROM   Students S, Enrolled E
WHERE  S.stdid = E.studentid AND E.grade = 'B'
```

Retrieve the names of the students who had a grade 'B' and the course in which they did so. Answer:

name	cid
Jones	CS101

Relational Query Languages (7)

Views as Named Queries

- A relation instance is normally defined extensionally (i.e., at each point in time we can enumerate the tuples that belong to it).
- A view defines a relation instance intensionally (i.e., by means of an expression that, when evaluated against a database instance, produces the corresponding relation instance).
- A view explicitly assigns a name to the relation it defines and implicitly characterizes its schema (given that the type of the expression can be inferred).
- Typically, (the substantive part of) the view definition language is the (D)QL,
- For a view, therefore, the DBMS only need store the query expression, not a set of tuples, as the latter can be obtained, whenever needed, by evaluating the former.

Relational Query Languages (8)

View Definition and Usage

- Start with:

Example

```
CREATE TopStudents (sname, stid, courseid)
    AS SELECT S.name, S.stdid, E.cid
       FROM   Students S, Enrolled E
       WHERE  S.stdid = E.studentid and E.grade = 'B'
```

- Follow up with:

Example

```
SELECT T.sname, T.courseid
FROM   TopStudents T
```

- This should look familiar.

Relational Query Languages (9)

Why are views useful?

- Views can be used to present necessary information (or a summary thereof), while hiding details in underlying relation(s).
- Views can be used to project specific abstractions to specific applications.
- Views can be materialized (e.g., in a data warehouse, to make OLAP feasible).
- Views are a useful mechanism for controlled fragmentation and integration in distributed environments.

Summary

Relational Model, Databases, Query Languages

- The relational model remains the best formal foundation for the study of DBMSs.
- It brings out the crucial role of query languages in providing convenient mechanisms for interacting with the data.
- It lies behind the most successful DBMSs used by organizations today.

Advanced Database Management Systems

A Relational Algebra

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Preliminaries

- A query is applied to relation instances.
- The result of a query is also a relation instance.
- The schemas of the input/argument relations of a query are fixed.
- The schema for the result of a given query is statically known by type inference on the schemas of the input/argument relations.
- Recall that relational algebra is **closed** (equiv., has a closure property), i.e., the input(s) and output of any relational-algebraic expression is a relation.

Relational Algebra (1)

Relation Instances Used in Examples (1)

- Let Sailors, Boats and Reservations be example relations.
- Their schemas are on the right.
- Underlined sets of fields denote a key.

Example

Sailors (sid: integer, sname: string, rating: integer, age: real)
Boats (bid: integer, bname: string, colour: string)
Reservations (sid: integer, bid: integer, day: date)

Relational Algebra (2)

Relation Instances Used in Examples (2)

- The various relation instances used on the right.
- Note that there are two relation instances (viz., S1 and S2) for Sailors, one for Reservations (viz., R1), and none, yet, for Boats.
- Fields in an instance of one of these relations are referred to by name or by position (in which case the order is that of appearance, left to right, in the schema).

Example

S1 =

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2 =

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

R1 =

sid	bid	day
22	101	10/10/96
58	103	12/11/96

Relational Algebra (3)

Primitive Operations

σ : **selection** returns those rows in the single argument relation that satisfy the given predicate

π : **projection** deletes those columns in the single argument relation that are not explicitly asked for

$\times$: **(Cartesian, or cross-) product** concatenates each row in the first argument relation with each row in the second to form a row in the output

$\setminus$: **(set) difference** returns the rows in the first argument relation that are not in the second

$\cup$: **(set) union** returns the rows that are either in the first or in the second argument relation (or in both)

The above is a complete set: any other relational-algebra can be derived by a combination of the above.

Relational Algebra (4)

Derived Operations

$\cap$: $R \cap S \Leftrightarrow (R \cup S) \setminus ((R \setminus S) \cup (S \setminus R))$
(set) intersection returns the rows that are both in the first and in the second argument relation

$\bowtie$: $R \bowtie_\theta S \Leftrightarrow \sigma_\theta(R \times S)$
join concatenates each row in the first argument relation with each row in the second and form with them a row in the output provided that it satisfies the given predicate

$\div$: (see below for derivation)
division returns the rows in the first argument relation that are associated with every row in the second argument relation

Relational Algebra (5)

Extensions (1)

Useful extensions for clarity of exposition are:

ρ : **renaming** returns the same relation instance passed as argument but assigns the given name(s) to the output relation (or any of its attributes)

$\leftarrow$: **assignment** assigns the left-hand side name to the relation instance resulting from evaluating the right-hand side expression

Relational Algebra (6)

Extensions (2)

Extensions that change the expressiveness of classical relation algebra include (see, e.g., [Silberschatz et al., 2005]):

- **generalized projection**, which allows arithmetic expressions (and not just attribute names) to be specified
- **(group-by) aggregation**, which allows functions (such as **count**, **sum**, **max**, **min**, **avg**) to be applied on some attribute (possibly over partitions defined by the given group-by attribute)
- other kinds of join (e.g., semijoin, antijoin, [left|right] outer join)

Selection

- $\sigma_\theta(R) = \{x \mid \exists x \in R\,(\theta(x))\}$
- Rows in the single input relation R that satisfy the given selection condition (i.e., a Boolean expression on the available attributes) are in the output relation O.
- No duplicate rows can appear in O, so the cardinality of O cannot be larger than that of R.
- The schema of O is identical to the schema of R.
- The arity of O is the same as that of R.

Example

$\sigma_{rating>8}(S2) =$

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$\sigma_{sid>10 \wedge age<45.0}(\sigma_{rating>8}(S2)) =$

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

Projection

- $\pi_{a_1,\ldots,a_n}(R) = \{y \mid \exists x \in R\,(y.a_1 = x.a_1 \wedge \ldots \wedge y.a_n = x.a_n)\}$
- Columns in the single input relation R that are not in the given projection list do not appear in the output relation O.
- Duplicate rows might appear in O unless they are explicitly removed, and, if so, the cardinality of O is the same as that of R.
- The schema of O maps one-to-one to the given projection list, so the arity of O cannot be larger than that of R.

Example

$\pi_{sname,rating}(S2) =$

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$\pi_{sname,rating}(\sigma_{rating>8}(S2)) =$

sname	rating
yuppy	9
rusty	10

Set Operations (1)

Union, Intersection, Difference

- These binary operations have the expected set-theoretic semantics.
- Both input arguments R and S must have compatible schemas (i.e., their arity must be the same and the columns have to have the same types one-to-one, left to right).
- The arity of O is identical to that of I.
- The cardinality of O may be larger than that of the largest between in R and S in the case of union, but not in the case of intersection and difference.

Example

$S1 \cup S2 =$

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

Set Operations (2)

Union, Intersection, Difference

Example

$S1 \cap S2$

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

Example

$S1 \setminus S2$

sid	sname	rating	age
22	dustin	7	45.0

Cartesian/Cross Product

- $R \times S = \{xy \mid \exists x \in R\, \exists y \in S\}$
- The schema of O is the concatenation of the schemas of R and S, unless there is a name clash, in which case renaming can be used.
- The arity of O is the sum of the arities of R and S.
- The cardinality of O is the product of the cardinalities of R and S.

Example

$\rho_{1 \rightarrow sid1,\, 5 \rightarrow sid2}(S1 \times R1) =$

sid1	sname	rating	age	sid2	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	12/11/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	12/11/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	12/11/96

Joins (1)

θ-Join

- $R \bowtie_\theta S = \{xy \mid \exists x \in R\, \exists y \in S\, (\theta(xy))\}$
- $R \bowtie_\theta S \equiv \sigma_\theta(R \times S)$
- The schema of O is as for Cartesian product, as is arity.
- The cardinality of O cannot be larger than the product of the cardinalities of R and S.

Example

$\rho_{1 \to sid1,\, 5 \to sid2}(S1 \bowtie_{S1.sid < R1.sid} R1) =$

sid1	sname	rating	age	sid2	bid	day
22	dustin	7	45.0	58	103	12/11/96
31	lubber	8	55.5	58	103	12/11/96

Joins (2)

Equijoin and Natural Join

- An **equijoin** is a *theta*-join in which all terms in θ are equalities.
- In an equijoin, the schema of O is as for Cartesian product but only one of the equated columns is projected out, so the arity reduces by one for each such case.
- A **natural join** is an **equijoin** on <u>all</u> common columns.
- One only needs list the common columns in the condition.

Example

$S1 \bowtie_{sid} R1 =$

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	12/11/96

Division (1)
Through Examples

- In integer division, given two integers A and B, $A \div B$ is the largest integer Q such that $Q \times B \leq A$.
- In relational division, given two relations R and S, $R \div S$ is the largest relation instance O such that $O \times S \subseteq R$.
- If R lists suppliers and parts they supply, and S parts, then $R \div S$ lists suppliers of all S parts.

Example

$A =$

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

Example

$B_1 =$

pno
p2

$B_2 =$

pno
p2
p4

$B_3 =$

pno
p1
p2
p4

Example

$A \div B_1 =$

sno
s1
s2
s3
s4

$A \div B_2 =$

sno
s1
s4

$A \div B_3 =$

sno
s1

Division (2)
Through Rewriting (1)

- Division, like join, can be defined by rewriting into primitive operations but, unlike join, it is not used very often, so most DBMSs do not implement special algorithms for it.
- The schema of O is the schema of R minus the columns shared with S, so the arity of O cannot be as large as that of R.
- The cardinality of O cannot be larger than that of R.

Division (3)
Through Rewriting (2)

- Abusing notation, we can define division in terms of primitive operators as follows.
- Let r and s be relations with schemas R and S, respectively, and let $S \subseteq R$, then:
  - $T_1 \leftarrow \pi_{R-S}(r) \times s$ computes the Cartesian product of $\pi_{R-S}(r)$ and s so that each tuple $t \in \pi_{R-S}(r)$ is paired with every s-tuple.
  - $T_2 \leftarrow \pi_{R-S,S}(r)$ merely reorders the attributes of r in preparation for the set operation to come.
  - $T_3 \leftarrow \pi_{R-S}(T_1 - T_2)$ only retains those tuples $t \in \pi_{R-S}(r)$ such that for some tuple u in s, $tu \notin r$.
  - $r \div s \leftarrow \pi_{R-S}(r) - T_3$ only retains those tuples $t \in \pi_{R-S}(r)$ such that for all tuples u in s, $tu \in r$.

Generalized Projection

- Recall that generalized projection allows arithmetic expressions involving attribute names (and not just the latter) in the projection list.
- The first example to the right returns the sailor names with their associated ranking tripled.
- There is a further extended version that allows concomitant renaming as shown in the second example to the right.

Example

$\pi_{sname, rating*3}(S2) =$

sname	rating
yuppy	27
lubber	24
guppy	15
rusty	30

$\pi_{sname, rating*3 \rightarrow triplerating}(S2) \equiv$
$\rho_{2 \rightarrow triplerating}(\pi_{sname, rating*2}(S2)) =$

sname	triplerating
yuppy	27
lubber	24
guppy	15
rusty	30

Aggregation

- Recall that aggregation reduces a set of values into a single values by the application of a function such as **count**, **sum**, **max**, **min** or **avg**.
- It is also possible to form groups by attribute values, e.g., to take the average rating by age.
- Concomitant renaming can also be used.

Example

$\gamma_{avg(age) \rightarrow averageage}(S2) =$

averageage
40.125

$_{age}\gamma_{avg(rating) \rightarrow averagerating}(S2) =$

age	averagerating
35.0	8
55.0	8

Example Relational Algebra Expressions (1)

More Relation Instances

- For the next batch of examples, the various relation instances used are as shown.

Example

S3 =

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5

Example

R2 =

sid	bid	day
22	101	10/10/98
22	102	10/10/98
22	103	08/10/98
22	104	07/10/98
31	102	10/11/98
31	103	06/11/98
31	104	12/11/98
64	101	05/09/98
64	102	08/09/98
64	103	08/09/98

B1 =

bid	bname	colour
101	interlake	blue
102	interlake	red
103	clipper	green
104	marine	red

Example Relational Algebra Expressions (2)

Find the names of the sailors who have reserved boat 103

- $O_1 = \pi_{sname}(\sigma_{bid=103}(Reservations \bowtie Sailors))$
- $O_2 = \pi_{sname}((\sigma_{bid=103}(Reservations)) \bowtie Sailors)$
- $O_1 \equiv O_2$

Example

sname
dustin
lubber
horatio

Example Relational Algebra Expressions (3)

Find the names of the sailors who have reserved a red boat

- Information about boat colour is only available in Boats, so an extra join is needed.
- $O_1 = \pi_{sname}(\sigma_{color=red}(Boats) \bowtie (Reservations \bowtie Sailors))$
- $O_2 = \pi_{sname}(\pi_{sid}(\pi_{bid}(\sigma_{color=red}(Boats)) \bowtie Reservations) \bowtie Sailors)$
- $O_1 \equiv O_2$

Example

sname
dustin
lubber
horatio

Example Relational Algebra Expressions (4)

Find the names of the sailors who have reserved a red or a green boat

- Using assignment:
  1. $T_1 \leftarrow (\sigma_{colour=red}(Boats) \cup \sigma_{colour=green}(Boats))$
  2. $O \leftarrow \pi_{sname}(T_1 \bowtie (Reservations \bowtie Sailors))$
- Or:
  1. $T_1 \leftarrow (\sigma_{colour=red \vee colour=green}(Boats)$
  2. $O \leftarrow \pi_{sname}(T_1 \bowtie (Reservations \bowtie Sailors))$

Example

sname
dustin
lubber
horatio

Example Relational Algebra Expressions (5)

Find the names of the sailors who have reserved a red and a green boat

- Replacing $\cup$ with $\cap$ in the previous example doesn't work.
- Using assignment:
  1. $T_1 \leftarrow \pi_{sid}(\sigma_{colour=red}(Boats) \bowtie Reservations)$
  2. $T_2 \leftarrow \pi_{sid}(\sigma_{colour=green}(Boats) \bowtie Reservations)$
  3. $O \leftarrow \pi_{sname}((T_1 \cap T_2) \bowtie Sailors)$
- On the other hand, $\pi_{sname}((T_1 \cup T_2) \bowtie Sailors)$ does work for the previous example.

Example

sname
dustin
lubber

Example Relational Algebra Expressions (6)

Find the ids of sailors older than 20 who have not reserved a red boat

- Using assignment:
  1. $T_1 \leftarrow \pi_{sid}(\sigma_{age>20}(Sailors)$
  2. $T_2 \leftarrow \pi_{sid}((\sigma_{colour=red}(Boats) \bowtie Reservations) \bowtie Sailors)$
  3. $O \leftarrow T_1 \setminus T_2$

Example

sid
29
32
58
74
85
95

Example Relational Algebra Expressions (7)

Find the names of sailors who have reserved all boats

- The word **all** suggests the need for division.
- Projections are essential to arrange the schemas appropriately.
- Joins may be needed to recover columns that had to be dropped.
-
  1. $T_1 \leftarrow \pi_{sid,bid}(Reservations) \div \pi_{bid}(Boats)$
  2. $O \leftarrow \pi_{sname}(T_1 \bowtie Sailors)$

Example

sname
dustin

Summary
Relational Algebra

- An algebra, often extending, or modelled on, the relational algebra lies at the heart of most advanced DBMSs.
- It is the most used target formalism for the internal representation of logical plans.
- Rewriting that is based on logical-algebraic equivalences is an important task in query optimization (as we will discuss later).

Advanced Database Management Systems
SQL

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Example Relation Instances Again

Syntax and Semantics

Example SQL Queries

More Syntax and Semantics

SQL
Relation Instances Used in Examples

- Recall the relation instances on the right.
- These have been used before and will be used again, as before, in the examples that follow.

Example

S1 =

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2 =

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

R1 =

sid	bid	day
22	101	10/10/96
58	103	12/11/96

Core, Informal SQL Syntax (1)

The SELECT, FROM and WHERE Clauses

Definition

SELECT [DISTINCT] ⟨ target list ⟩
FROM ⟨ relation list ⟩
WHERE ⟨ qualification ⟩

- The SELECT clause defines which columns participate in the result, i.e., it plays the role of the relational-algebraic π operation.
- The FROM clause defines which relations are used as inputs, i.e., it corresponds to the leaves in a relational-algebraic expression.
- The WHERE clause defines the (possibly complex) predicate expression which a row must satisfy to participate in the result, i.e., it supplies the predicates for relational-algebraic operations like σ and $\bowtie$.
- DISTINCT is an optional keyword indicating that duplicates must be removed from the answer.

Syntax and Semantics

Core, Informal SQL Syntax (2)

The Arguments to SELECT, FROM and WHERE Clauses

- The argument to a FROM clause is a list of relation names (possibly with a range variable after each name, which allows a row in that relation to referred to elsewhere in the query).
- It is good practice to use range variables, so use them.
- The argument to a SELECT clause is a list of expressions based on attributes taken from the relations in the **relation list**.
- If '*' is used instead of a SELECT list, all attributes of are selected.
- The argument to a WHERE clause is referred to as a **qualification**, i.e., a Boolean expression whose terms are comparisons (of the form $E\ op\ const$ or $E_1\ op\ E_2$ where E, E_1, E_2 are, typically, attributes taken from the relations in the **relation list**, and $op \in \{<, >, =, >=, =<, <>\}$) combined using the connectives AND, OR and NOT.

Core, Informal SQL Semantics (1)

Three-to-Four Steps to the Answer

- To characterize what is the answer to a SQL query:
  1. Compute the cross-product of relations in the FROM list, call it J.
  2. Discard tuples in J that fail the qualification, call the result S.
  3. Delete from S any attribute that is not in the SELECT list, call the result P.
  4. If DISTINCT is specified, eliminate duplicate rows in P to obtain the result A, otherwise A=P.
- While as an evaluation strategy, the procedure above is likely to be very inefficient, it provides a simple, clear characterization of the answer to a query.
- As we will see, an optimizer is likely to find more efficient evaluation strategies to compute the same answer.

Core, Informal SQL Semantics (2)

Find the names of the sailors who have reserved boat 103.

Example

```
SELECT   S.sname
FROM     Sailors S, Reservations R
WHERE    S.sid = R.sid AND R.bid = 103
```

Example

Assume the database state contains $\{S1, R1\}$. Then, in Step 1, $S1 \times R1 =$

sid1	sname	rating	age	sid2	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	12/11/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	12/11/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	12/11/96

In Step 2, $\sigma_{S.sid=R.sid \wedge R.bid=103}(S1 \times R1)$

sid1	sname	rating	age	sid2	bid	day
58	rusty	10	35.0	58	103	12/11/96

In Step 3, $\pi_{sname}(\sigma_{S.sid=R.sid \wedge R.bid=103}(S1 \times R1))$

sname
rusty

No DISTINCT implies no Step 4.

Core, Informal SQL Semantics (3)

Find the ids of sailors who have reserved at least one boat

Example

```
SELECT   S.sid
FROM     Sailors S, Reservations R
WHERE    S.sid = R.sid
```

Example

Assume the database state contains $\{S1, R1\}$. Then, Step 1, $S1 \times R1$ produces the same results as in the previous example.

In Step 2, $\sigma_{S.sid=R.sid}(S1 \times R1)$

sid1	sname	rating	age	sid2	bid	day
22	dustin	7	45.0	22	101	10/10/96
58	rusty	10	35.0	58	103	12/11/96

In Step 3, $\pi_{sid}(\sigma_{S.sid=R.sid}(S1 \times R1))$

sid
22
58

No DISTINCT implies no Step 4.

Core, Informal SQL Syntax (3)

Expressions and Strings

- Arithmetic expressions and string pattern matching can also be used.
- AS and = are two ways to name fields in result.
- LIKE is used for string matching. '_' stands for any one character and '%' stands for zero or more arbitrary characters.

Example

```
SELECT   S.age,
         age1=S.age-5,
         2*S.age AS age2
FROM     Sailors S
WHERE    S.sname LIKE 'Y_%Y'
```

Over S2, the answer would be:

age	age1	age 2
35.0	30.0	70.0

Example SQL Queries (1)

More Relation Instances

- For the next batch of examples, the various relation instances used are also known from previous examples.

Example

S3 =

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5

Example

R2 =

sid	bid	day
22	101	10/10/98
22	102	10/10/98
22	103	08/10/98
22	104	07/10/98
31	102	10/11/98
31	103	06/11/98
31	104	12/11/98
64	101	05/09/98
64	102	08/09/98
64	103	08/09/98

B1 =

bid	bname	colour
101	interlake	blue
102	interlake	red
103	clipper	green
104	marine	red

Example SQL Queries (2)

Find the names of the sailors who have reserved a red boat

- Recall $\pi_{sname}(\sigma_{color=red}(Boats) \bowtie (Reservations \bowtie Sailors))$

Example

```
SELECT  S.sname
FROM    Sailors S, Boats B, Reservations R
WHERE   S.sid = R.sid
        AND R.bid = B.bid
        AND B.colour = 'red'
```

Example SQL Queries (3)

Find the sids of the sailors who have reserved a red or a green boat

Example

Either:
```
SELECT  S.sid
FROM    Sailors S, Boats B, Reservations R
WHERE   S.sid = R.sid
        AND R.bid = B.bid
        AND (B.colour = 'red' OR B.colour = 'green')
```

Or:
```
SELECT  S1.sid
FROM    Sailors S1, Boats B1, Reservations R1
WHERE   S1.sid = R1.sid
        AND R1.bid = B1.bid
        AND B1.colour = 'red'
UNION
SELECT  S2.sid
FROM    Sailors S2, Boats B2, Reservations R2
WHERE   S2.sid = R2.sid
        AND R2.bid = B2.bid
        AND B2.colour = 'green'
```

Set difference is captured by EXCEPT, e.g., to find the sids of the sailors who have reserved a red but not a green boat.

Example SQL Queries (4)

Beware being quick when there are quirks

Example

Contrast this query:
```
SELECT  S.sid
FROM    Sailors S, Boats B, Reservations R
WHERE   S.sid = R.sid
        AND R.bid = B.bid
        AND (B.colour = 'red' AND B.colour = 'green')
```

with this one:
```
SELECT     S.sid
FROM       Sailors S, Boats B, Reservations R
WHERE      S.sid = R.sid
           AND R.bid = B.bid
           AND B.colour = 'red'
INTERSECT
SELECT     S.sid
FROM       Sailors S, Boats B, Reservations R
WHERE      S.sid = R.sid
           AND R.bid = B.bid
           AND B.colour = 'green'
```

and this one:
```
SELECT  S.sid
FROM    Sailors S, Boats B1, Reservations R1,
        Boats B2, Reservations R2,
WHERE   S.sid = R1.sid AND S.sid=R2.Sid
        AND R1.bid = B1.bid AND R2.Bid = B2.Bid
        AND (B1.colour = 'red' AND B2.colour = 'green')
```

Core, Informal SQL Syntax (4)
Nested Queries

Example

```
SELECT   S.sname
FROM     Sailors S
WHERE    S.sid IN (  SELECT   R.sid
                     FROM     Reservations R
                     WHERE    R.bid = 103
         )
```

- The ability to nest queries is a powerful feature of SQL.
- Queries can be nested in the WHERE, FROM and HAVING clauses.
- To understand the semantics of nested queries, think of a nested loop, i.e., for each Sailors tuple, compute the nested query and which pass the IN qualification.

Core, Informal SQL Syntax (5)
Correlated Nested Queries

- We could have used NOT IN to find sailor who have not reserved boat 103.
- Another set comparison operator (implicitly, with the empty set) is EXISTS, and see the Bibliography for yet more.
- It is also possible to correlate the queries via shared range variables.

Example

```
SELECT   S.sname
FROM     Sailors S
WHERE    EXISTS (  SELECT   *
                   FROM     Reservations R
                   WHERE    R.bid = 103 AND R.sid = S.sid
         )
```

Core, Informal SQL Syntax (6)

Aggregate Operators

Definition

COUNT	([DISTINCT] A)	the number of (unique) values in the A column
SUM	([DISTINCT] A)	the sum of all (unique) values in the A column
AVG	([DISTINCT] A)	the average of all (unique) values in the A column
MAX	(A)	the maximum value in the A column
MIN	(A)	the minimum value in the A column

- Aggregate operators are another significant extension of relational algebra in SQL.
- They take a collection of values as input and return a single value as output.

Example SQL Queries (5)

Aggregation Queries

Example

- ```
  SELECT  COUNT(*)
  FROM    Sailors S
  ```
- ```
  SELECT  AVG(S.age)
  FROM    Sailors S
  WHERE   S.rating = 10
  ```
- ```
  SELECT  COUNT(DISTINCT S.rating)
  FROM    Sailors S
  WHERE   S.name = 'horatio' OR S.name = 'dusting'
  ```
- ```
  SELECT  S.sname
  FROM    Sailors S
  WHERE   S.rating = ( SELECT  MAX(S2.rating)
                       FROM    Sailors S2
                       WHERE   S2.sname = 'horatio'
                     )
  ```

Example SQL Queries (6)

Find the name and the age of the oldest sailor(s)

Example

- ```
  SELECT  S.sname, MAX(S.age) -- Illegal SQL!
  FROM    Sailors S
  ```
- ```
  SELECT  S.sname, S.age
  FROM    Sailors S
  WHERE   S.age = ( SELECT  MAX(S2.age)
                    FROM    Sailors S2
                  )
  ```

- The first query is illegal: if a SELECT clause uses an aggregate operation either it must do so for all attributes in the clause or else it must contain a GROUP BY clause.
- The second query, with a nested query, is legal and correct.

Core, Informal SQL Syntax (7)

Partitioned Aggregation

Definition

```
SELECT     [DISTINCT] ⟨ target list ⟩
FROM       ⟨ relation list ⟩
WHERE      ⟨ qualification ⟩
GROUP BY   ⟨ grouping list ⟩
HAVING     ⟨ grouping qualification ⟩
```

- A group is a partition of rows that agree on the values of the attributes in the grouping list.
- We can mix attribute names and applications of aggregate operations in the target list but the attribute names must be a subset of the grouping list, so that each row in the result corresponds to one group.
- The grouping qualification determines whether a row is produced in the answer for a given group.

Core, Informal SQL Semantics (4)

Three More Steps Than Before to the Answer

- To characterize what is the answer to this extended form of an SQL query:
  1. Compute the cross-product of relations in the FROM list, call it J.
  2. Discard tuples in J that fail the qualification, call the result S.
  3. Delete from S any attribute that is not in the SELECT list, call the result P.
  4. Sort P into groups by the value of attributes in the GROUP BY list, call the result G.
  5. Discard groups in G that fail the grouping qualification, call the result H.
  6. Generate one answer tuple per qualifying group, call the result T.
  7. If DISTINCT is specified, eliminate duplicate rows in T to obtain the result A, otherwise A=T.

Core, Informal SQL Semantics (5)

Find the age of the youngest sailor that is at least 18, for each rating with at least 2 such sailors

Example

```
SELECT    S.rating, MIN(S.age) AS minage
FROM      Sailors S
WHERE     S.age >= 18
GROUP BY  S.rating
HAVING    COUNT(*) >= 2
```

Assume the database state contains $\{S3\}$. Then, in Step 1, $S3 =$

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5

Example

After Steps 2 and 3, we have:

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5

Core, Informal SQL Semantics (6)

Find the age of the youngest sailor that is at least 18, for each rating with at least 2 such sailors

Example

After Step 4, we have:

rating	age
1	33.0
3	25.5
3	63.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

Example

After Step 5, we have:

rating	age
3	25.5
3	63.5
7	45.0
7	35.0
8	55.5
8	25.5

After Steps 6 and 7, we have:

rating	minage
3	25.5
7	35.0
8	25.5

Example SQL Queries (7)

For each red boat, find the number of reservations for this boat

Example

```
SELECT     B.bid, COUNT(*) AS reservationCount
FROM       Boats B, Reservations R
WHERE      R,bid = B.bid AND B.colour = 'red'
GROUP BY   B.Bid
```

Null Values

- Field values in a tuple are sometimes unknown (e.g., a rating has not been assigned) or inapplicable (e.g., no spouse's name).
- SQL provides a special value null for such situations.
- The presence of null complicates many issues.
  - Special operators are needed to check if value is/is not null.
  - Is `rating` $>$ 8 true or false when `rating` is equal to null? What about `AND`, `OR` and `NOT` connectives?
  - We need a 3-valued logic (true, false and unknown).
  - The meaning of many constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)
  - New operators (in particular, outer joins) are possible/needed.
- Null values can be disallowed when columns are defined.

Summary
SQL

- SQL was an important factor in the early acceptance of the relational model because it is more natural than earlier, procedural query languages.
- SQL is relationally complete (in fact, it has significantly more expressive power than relational algebra).
- Even queries that can be expressed in RA can often be expressed more naturally in SQL.
- There are usually many alternative ways to write a query, so an optimizer is needed to find an efficient evaluation plan.
- In practice, users need to be aware of how queries are optimized and evaluated for best results.
- NULL for unknown field values brings many complications

Advanced Database Management Systems

Query Processing: Logical Optimization

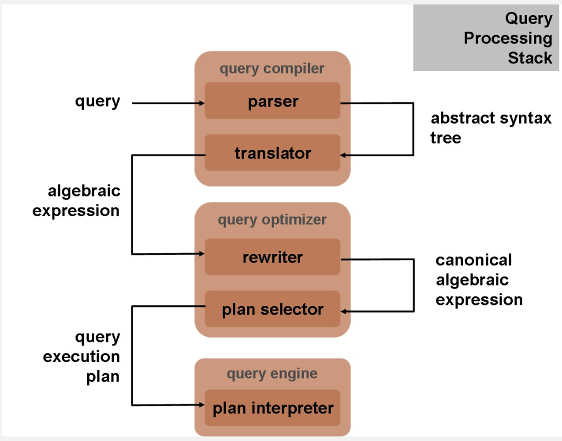
Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Overview of Query Processing (1)
The Query Processing Stack

1. **Parse** the declarative query
2. **Translate** to obtain an algebraic expression
3. **Rewrite** into a canonical, heuristically-efficient logical query execution plan (QEP)
4. **Select** the algorithms and access methods to obtain a quasi-optimal, costed concrete QEP
5. **Execute** (the typically interpretable form of) the procedural QEP



Overview of Query Processing (2)
Example Relations

- Let Flights and UsedFor be example relations.
- Flights asserts which flight number departs from where to where and its departure and arrival times.
- UsedFor asserts which plane is used for each flight on which weekday, Usable asserts which plane type (e.g., a 767) can be used for which flight, Certified asserts which pilot can fly which plane type.
- Their schemas are on the right.
- Underlined sets of fields denote a key.

Example (Schemas)

Flights (fltno: string, from: string, to: string, dep: date, arr: date)
UsedFor (planeid:string, fltid: string, weekday: string)
Usable (flid:string, pltype: string)
Certified (pilid:string, planetype: string)

Overview of Query Processing (3)

Example SQL Query and Corresponding Translator Output

Example (SQL)

```
SELECT   F.fltno, F.from
FROM     Flights F, UsedFor U
WHERE    F.fltno = U.fltid
```

Example (Algebraic Expression)

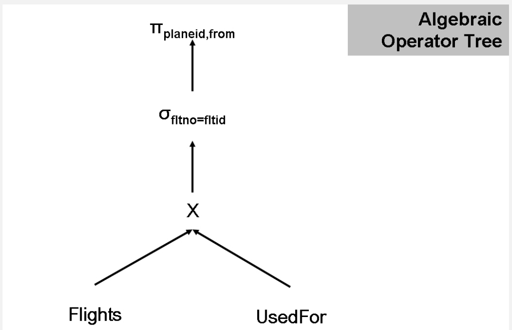
$\pi_{F.fltno, F.from}(\sigma_{F.fltno = U.fltid}(Flights \times UsedFor))$

Overview of Query Processing (4)

Algebraic Expression and Corresponding Algebraic Operator Tree

Example (Algebraic Expression)

$\pi_{F.fltno, F.from}(\sigma_{F.fltno = U.fltid}(Flights \times UsedFor))$

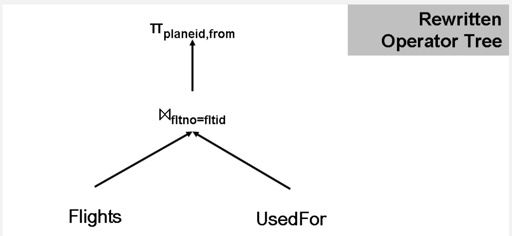


Overview of Query Processing (5)

Example Rewriting Rule and Corresponding Outcome

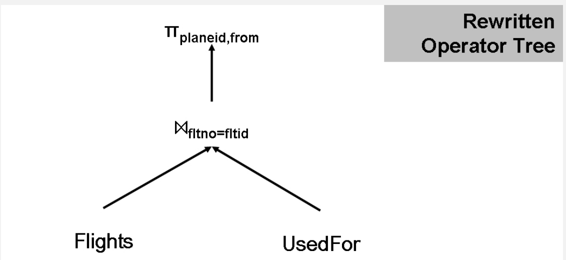
Example (Join Insertion Rule)

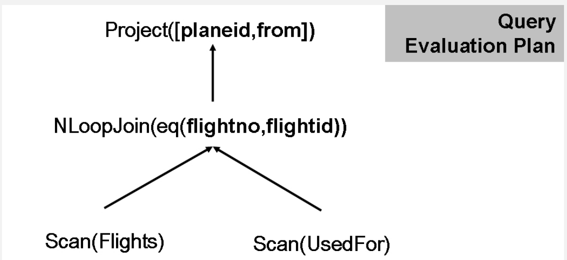
$\sigma_\theta(R \times S) \Leftrightarrow (R \bowtie_\theta S)$



Overview of Query Processing (6)

Rewritten Operator Tree and Corresponding QEP





Overview of Query Processing (7)

Translate, then Rewrite

- The outcome of translation is a relational-algebraic expression derived from a direct, clause-by-clause translation.
- A relational-algebraic expression can be represented as an algebraic operator tree.
- By applying rewrite rules, the (usually naive) algebraic operator tree can be rewritten into a heuristically-efficient canonical form, often called the **logical** QEP for the query.

Overview of Query Processing (8)

Compute Costs to Select the Algorithms, then Evaluate

- Typically, every algebraic operator can be implemented by different concrete algorithms.
- Using cost models, the plan selector considers which concrete algorithm to use for each operator in the logical QEP.
- The various concrete algorithms that implement each operator often adopt an iterator pattern.
- The result of this process is a **physical** QEP, i.e., one that expresses a concrete computational process in the actual environment in which the query is to be evaluated.
- The nodes (i.e., the selected concrete algorithms) in a physical QEP are often referred to as **physical operators**.
- The physical QEP can be compiled into an executable or, more commonly, remains an interpretable structure that a **query evaluation engine** knows how to process.

Overview of Query Processing (9)

The Big Questions

- There are three main issues in query optimization:
    1. **For a given algebraic expression, which rewrite rules to use to canonize it?** This determines what heuristic optimization decisions are carried out.
    2. **For a given canonical algebraic expression, which different concrete algorithm assignments to consider?** This determines the search space for finding the desired QEP.
    3. **If the desired plan is the one that results in the shortest response time, what cost models should be used to estimate the response time of a QEP?** This determines the (necessarily sub-optimal) choice of which QEP to use to evaluate the query.
- The above sequence of questions (with the strategy they imply) was first proposed in 1979 in IBM's System R, the first practical relational DBMS, and remains the dominant paradigm.

Relational-Algebraic Equivalences (1)

Logical Optimization (1)

- Logical optimization involves the transformation of an expression E in a language L into an equivalent expression E' also in L where E' is likely to admit of a more efficient evaluation than E.
- Transformation often are expressed as rewrite rules that express relational-algebraic equivalences of various kinds.
- The different rewrite rules have different purposes, among which:
    - to break down complex predicates in selections and long attribute lists in projections in order to enable more rewrites;
    - to move selections (which are cardinality-reducing) and projections (which are arity-reducing) upstream (i.e., towards the leaves) and thereby reduce the data volumes that downstream operators must contend with;
    - to enable entire subtrees to be implemented by a single efficient algorithm.

Relational-Algebraic Equivalences (1)

Logical Optimization (2)

- Recall that an **associative law** states that two applications of an operation ω can be performed in either order:
  $(x \ \omega \ y) \ \omega \ z \Leftrightarrow x \ \omega \ (y \ \omega \ z)$
- Recall that a **commutative law** states that the result an operation ω is independent of the order of the operands:
  $(x \ \omega \ y) \Leftrightarrow (y \ \omega \ x)$
- For example:
  1. $+$ is both commutative and associative.
  2. $-$ is neither.

Relational-Algebraic Equivalences (2)

Primitive/Derived Transformations

- The definitions of derived operations in terms of primitive ones are relational-algebraic equivalences.
- The most widely used among these (call it **R0** now, we have alluded to it already as a join-insertion rule) rewrites a Cartesian product followed by a selection into a join if the selection condition is a join condition, i.e.:

$$\sigma_\theta(R \times S) \quad \Leftrightarrow \quad (R \bowtie_\theta S)$$

Relational-Algebraic Equivalences (3)

Selection

- **R1**: A conjunctive selection condition can be broken up into a cascade (i.e., a sequence) of individual σ operations, i.e.:

$$\sigma_{\theta_1 \wedge \ldots \wedge \theta_n}(R) \quad \Leftrightarrow \quad \sigma_{\theta_1}(\ldots(\sigma_{\theta_n}(R))\ldots)$$

- **R2**: The σ operation is commutative, i.e.:

$$\sigma_{\theta_1}(\sigma_{\theta_2}(R)) \quad \Leftrightarrow \quad \sigma_{\theta_2}(\sigma_{\theta_1}(R))$$

Relational-Algebraic Equivalences (4)

Projection

- **R3**: In a cascade (i.e., a sequence) of individual π operations all but the last one can be ignored, i.e.:

$$\pi_{L_1}(\ldots(\pi_{L_n}(R))\ldots) \quad \Leftrightarrow \quad \pi_{L_1}(R)$$

- **R4**: The σ and π operations commute if the selection condition θ only involves attributes in the projection list $a_1, \ldots, a_n$, i.e.:

$$\pi_{a_1,\ldots,a_n}(\sigma_\theta(R)) \quad \Leftrightarrow \quad \sigma_\theta(\pi_{a_1,\ldots,a_n}(R))$$

Relational-Algebraic Equivalences (5)

Commutativity of Join and Cartesian Product (1)

- **R5**: Both the $\bowtie$ and the $\times$ operations are commutative, i.e.:

$$\begin{aligned} R \bowtie_\theta S &\Leftrightarrow S \bowtie_\theta R \\ R \times S &\Leftrightarrow S \times R \end{aligned}$$

- **R6.1**: The σ and $\bowtie$ (resp., $\times$) operations commute if the selection condition θ only involves attributes in one of the operands, i.e.:

$$\sigma_\theta(R \bowtie S) \Leftrightarrow \sigma_\theta(R) \bowtie S$$

- **R6.2**: The σ and $\bowtie$ (resp., $\times$) operations commute if the selection condition θ is of the form $\theta_1 \wedge \theta_2$ and θ_1 only involves attributes in one operand and θ_2 only involves attributes in the other, i.e.:

$$\sigma_\theta(R \bowtie S) \Leftrightarrow \sigma_{\theta_1}(R) \bowtie \sigma_{\theta_2}(S)$$

Relational-Algebraic Equivalences (6)

Commutativity of Join and Cartesian Product (2)

- **R7.1**: The π and $\bowtie$ (resp., $\times$) operations commute if the projection list is of the form $L = a_1, \ldots, a_m, b_1, \ldots, b_n$, the a_i are attributes of one operand, the b_j are attributes of the other, and the join condition θ only involves attributes in L, i.e.:

$$\pi_L(R \bowtie_\theta S) \Leftrightarrow (\pi_{a_1,\ldots,a_m}(R)) \bowtie_\theta (\pi_{b_1,\ldots,b_n}(S))$$

- **R7.2**: If the join condition θ includes R-attributes $a_{m+1}, \ldots, a_{m+k}$and $S-$attributes $b_{n+1}, \ldots, b_{n+l}$ not in L, then they need also to be projected from R and S and a final projection of L is still required, i.e.:

$$\pi_L(R \bowtie_\theta S) \Leftrightarrow \pi_L(\pi_{a_1,\ldots,a_m,a_{m+1},\ldots,a_{m+k}}(R)) \bowtie_\theta (\pi_{b_1,\ldots,b_n,b_{n+1},\ldots,b_{n+l}}(S))$$

- **R7.3**: Since there is no predicate in a Cartesian product **R7.1** always applies with $\bowtie_\theta$ replaced with $\times$.

Relational-Algebraic Equivalences (7)

Commutativity of Set Operations (1)

- **R8**: Both the $\cup$ and the $\cap$ (but **not** the $\setminus$) operations are commutative, i.e.:

$$\begin{array}{rcl} R \cup S & \Leftrightarrow & S \cup R \\ R \cap S & \Leftrightarrow & S \cap R \end{array}$$

- **R9**: The $\bowtie$, $\times$, $\cup$ and $\cap$ operations are individually associative, i.e.:

$$\begin{array}{rcl} (R \bowtie S) \bowtie T & \Leftrightarrow & R \bowtie (S \bowtie T) \\ (R \times S) \times T & \Leftrightarrow & R \times (S \times T) \\ (R \cup S) \cup T & \Leftrightarrow & R \cup (S \cup T) \\ (R \cap S) \cap T & \Leftrightarrow & R \cap (S \cap T) \end{array}$$

Relational-Algebraic Equivalences (8)

Commutativity of Set Operations (2)

- **R10**: The σ operation commutes with the $\cup$, $\cap$, **and** $\setminus$ operations, i.e.:

$$\begin{array}{rcl} \sigma_\theta(R \cup S) & \Leftrightarrow & (\sigma_\theta(R)) \cup (\sigma_\theta(S)) \\ \sigma_\theta(R \cap S) & \Leftrightarrow & (\sigma_\theta(R)) \cap (\sigma_\theta(S)) \\ \sigma_\theta(R \setminus S) & \Leftrightarrow & (\sigma_\theta(R)) \setminus (\sigma_\theta(S)) \end{array}$$

- **R11**: The π operation commutes with the $\cup$ operation, i.e.:

$$\pi_L(R \cup S) \quad \Leftrightarrow \quad (\pi_L(R) \cup \pi_L(S))$$

Relational-Algebraic Equivalences (9)

Other Transformations

- Predicates can also be rewritten (e.g., using DeMorgan's laws to push negation into conjuncts and disjuncts).
- Note that using **R1** to rewrite separate predicates into a conjunction thereof could, depending on the compiler, lead to a contradiction, in which case the selection would be satisfied by no tuple in the input, resulting in an empty result.

A Heuristic Algebraic Optimization Strategy (1)

1. Use **R1** to break up complex select conditions into individual ones. This creates opportunities for pushing selections down towards the leaves, thereby reducing the cardinality of intermediate results.
2. Use **R2**, **R4**, **R6** and **R10**, which define commutativity for select and pushing selections down towards the leaves.
3. Use **R5**, **R8** and **R9**, which define commutativity and associativity for binary operations, to rearrange the leaf nodes. The following criteria can be used:
   3.1 Ensure that the leaves under the most restrictive selections (i.e., those that reduce the size of the result) are positioned to execute first (typically, as the left operand). This can rely on selectivity estimates computed from metadata in the system catalogue, as we shall see.
   3.2 Avoid causing Cartesian products to be used, i.e., override the above criterion if it would not lead to a join being placed above the operands.

A Heuristic Algebraic Optimization Strategy (2)

4. Use **R0** to combine selections on the result of Cartesian products into a join. This allows efficient join algorithms to be used
5. Use **R3**, **R4**, **R7** and **R11**, which define how project cascades and commutes with other operations, to break up and move projection lists down towards the leaves, thereby reducing the arity of intermediate results. Only those attributes needed downstream in the plan need be kept from any operation.
6. Identify subtrees that express a composition of operations for which there is available a single, specific algorithm that computes the result of the composition.

An Example Run (1)

Schemas and Query

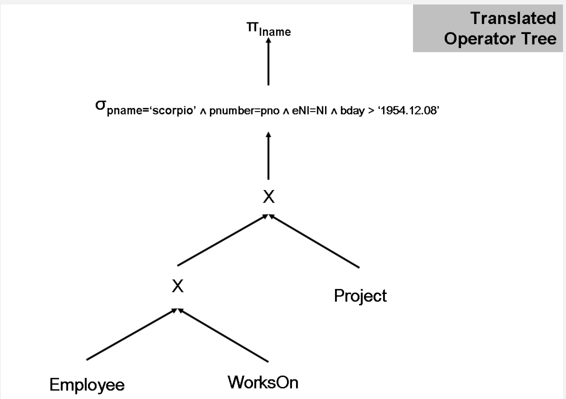
Example (Simplified Schema)

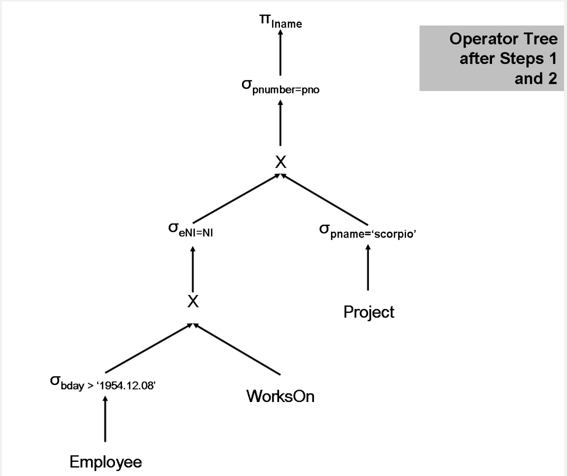
Employee (fname, mname, lname, bday, address, sex, sal, <u>NI</u>, dno)
Project (pname, <u>pnumber</u>, ploc, dnum)
WorksOn (<u>eNI, pno</u>, hours)

Example (SQL Query)

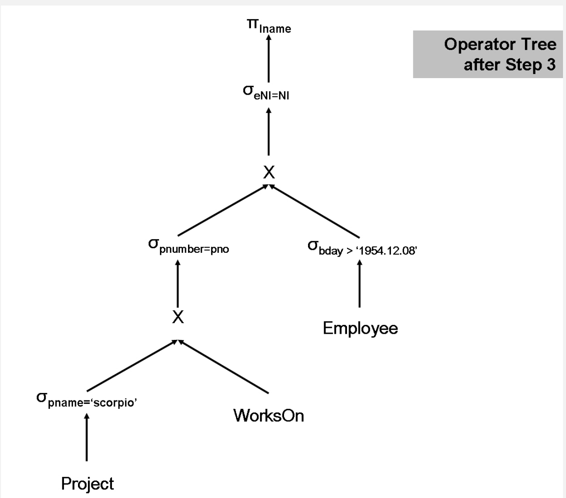
```
SELECT E.lname
FROM   Employee E, WorksOn W, Project P
WHERE  P.pname = 'Scorpio'
  AND  P.pnumber = W.pno AND W.eNI = e.NI
  AND  E.bday > '1954.12.08'
```

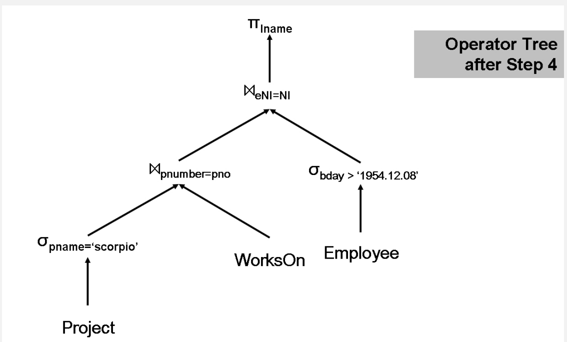
An Example Run (2)



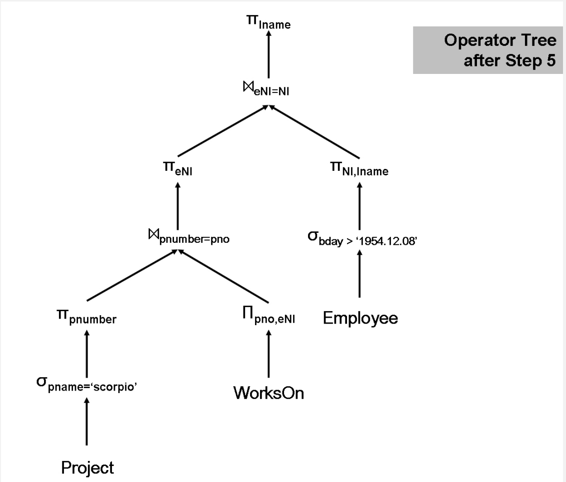


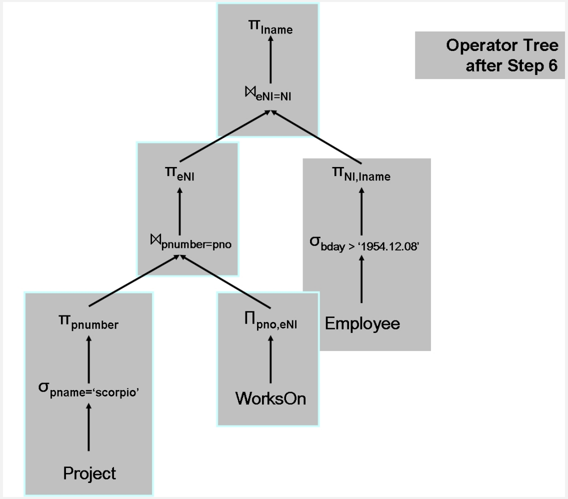
An Example Run (3)





An Example Run (4)





Summary

Equivalence-Based Rewriting of Logical QEPs

- For a given algebraic expression, deciding which rewrite rules to use to canonize it is a major issue in query processing.
- This determines what heuristic optimization decisions are carried out.
- Equivalence-based rewriting of logical QEPs is a classical strategy.
- It is sufficiently well-established and well-understood for there to be a consensus on what works fairly well in the classical cases.
- There is great uncertainty as to what extent rewriting is also useful in several kinds of advanced DBMSs.
- In any case, cost-based optimization is always fundamentally required.

Acknowledgements

The material presented mixes original material by the author and by Norman Paton as well as material adapted from

- [Elmasri and Navathe, 2006]
- [Garcia-Molina et al., 2002]
- [Ramakrishnan and Gehrke, 2003]
- [Silberschatz et al., 2005]

The author gratefully acknowledges the work of the authors cited while assuming complete responsibility any for mistake introduced in the adaptation of the material.

References

Elmasri, R. and Navathe, S. B. (2006).
Fundamentals of Database Systems.
Addison-Wesley, 5^{th} edition.

Garcia-Molina, H., Ullman, J. D., and Widom, J. (2002).
Database Systems: The Complete Book.
Pearson Education Limited, 1^{st} edition.

Ramakrishnan, R. and Gehrke, J. (2003).
Database Management Systems.
McGraw-Hill Education - Europe, 3^{rd} edition.

Silberschatz, A., Korth, H. F., and Sudarshan, S. (2005).
Database System Concepts.
McGraw-Hill Education - Europe, 5^{th} edition.