Relational Databases

Jan Chomicki
University at Buffalo
Plan of the course

1. Relational databases
2. Relational database design
3. Conceptual database design
4. Object databases
5. XML databases
6. Advanced topics
Part I

Relational data model
Outline of Part I

1. Basic concepts

2. Integrity constraints
Relational data model
Relational data model

Domain

- **domain**: predefined set of atomic values: integers, strings,…
- every attribute value comes from a domain or is null (null is not a value)
- **First Normal Form**: domains consist of atomic values
### Relational data model

#### Domain
- **domain**: predefined set of atomic values: integers, strings,...
- every attribute value comes from a domain or is null (null is not a value)
- **First Normal Form**: domains consist of atomic values

#### Tuple (row)
- **tuple**: a sequence of values and nulls
- **tuple arity**: the number of values in the sequence (including nulls)
Relational data model

Domain
- **domain**: predefined set of atomic values: integers, strings,...
- Every attribute value comes from a domain or is null (null is not a value)
- **First Normal Form**: domains consist of atomic values

Tuple (row)
- **tuple**: a sequence of values and nulls
- **tuple arity**: the number of values in the sequence (including nulls)

Relation
- **relation name**, e.g., Employee
- **relation schema**: finite set of attributes (column labels) and associated domains, for example
  - Name:String, Salary:Decimal, Age:Integer
- **relation instance**: finite set of tuples conforming to the schema.
Schema vs. instance

Schema
- rarely changes
- when it does, database needs to be reorganized
- used to formulate queries

Instance
- changes with update transactions
- used to evaluate queries

Notation
- An instance of a schema $R$ is denoted as $r$.

We will need the schema vs. instance distinction in discussing integrity constraints and query results.
Schema vs. instance

**Schema**
- rarely changes
- when it does, database needs to be reorganized
- used to **formulate** queries

An instance of a schema $R$ is denoted $r$. We will need the schema vs. instance distinction in discussing integrity constraints and query results.
Schema vs. instance

Schema
- rarely changes
- when it does, database needs to be reorganized
- used to formulate queries

Instance
- changes with update transactions
- used to evaluate queries

Notation
An instance of a schema $R$ is denoted $r$. 
Schema vs. instance

Schema
- rarely changes
- when it does, database needs to be reorganized
- used to formulate queries

Instance
- changes with update transactions
- used to evaluate queries

Notation
An instance of a schema $R$ is denoted $r$.

We will need the schema vs. instance distinction in discussing integrity constraints and query results.
Integrity constraints

Logical conditions that have to be satisfied in every database instance.
Integrity constraints

Logical conditions that have to be satisfied in every database instance.

### Role of constraints

- **guarding** against entering incorrect data into a database (**data quality**)
- providing **object identity** (key and foreign key constraints)
- representing relationships and associations
- helping in **database design**
Integrity constraints

Logical conditions that have to be satisfied in every database instance.

Role of constraints

- guarding against entering incorrect data into a database (data quality)
- providing object identity (key and foreign key constraints)
- representing relationships and associations
- helping in database design

DBMS support for constraints

- all declared constraints are checked after every transaction
- if any constraint is violated, the transaction is backed out
- typically SQL DBMS support only limited kinds of constraints
  - keys, foreign keys, CHECK constraints
Key constraints

A key constraint of a relation schema $R$ is a set of attributes $S$ (called a key) of $R$. An instance $r$ satisfies a key constraint $S$ if $r$ does not contain a pair of tuples that agree on $S$ but disagree on some other attribute of $R$. Formally: for each two tuples $t_1 \in r$, $t_2 \in r$ if $t_1[S] = t_2[S]$, then $t_1[A] = t_2[A]$ for every attribute $A$ in $R$. 

Jan Chomicki ()
Relational databases
Key constraints

**Key constraint of a relation schema** $R$

A set of attributes $S$ (called a *key*) of $R$. 

Formally: for each two tuples $t_1 \in r$, $t_2 \in r$ if $t_1[S] = t_2[S]$, then $t_1[A] = t_2[A]$ for every attribute $A$ in $R$. 
Key constraints

Key constraint of a relation schema $R$

A set of attributes $S$ (called a key) of $R$.

An instance $r$ satisfies a key constraint $S$ if $r$ does not contain a pair of tuples that agree on $S$ but disagree on some other attribute of $R$. 
Key constraints

Key constraint of a relation schema \( R \)
A set of attributes \( S \) (called a **key**) of \( R \).

An instance \( r \) **satisfies** a key constraint \( S \) if \( r \) does not contain a pair of tuples that agree on \( S \) but disagree on some other attribute of \( R \).

Formally: for each two tuples \( t_1 \in r, t_2 \in r \)

\[ \text{if } t_1[S] = t_2[S], \text{ then } t_1[A] = t_2[A] \text{ for every attribute } A \text{ in } R. \]
Properties of keys

Adequacy

Uniqueness of key values should be guaranteed by the properties of the application domain in other words: it is an error to have different tuples (in the same relation) with the same key values.

A key should be as small as possible (good database design).

Minimality

No subset of a key can also be designated a key.

Multiple keys

There may be more than one key in a relation schema.

One is selected as the primary key:

- Cannot be null (entity integrity)
- Typically used in indexing
Properties of keys

Adequacy

- uniqueness of key values should be guaranteed by the properties of the application domain
- in other words: it is an error to have different tuples (in the same relation) with the same key values
- a key should be as small as possible (good database design)
Properties of keys

Adequacy
- uniqueness of key values should be guaranteed by the properties of the application domain
- in other words: it is an error to have different tuples (in the same relation) with the same key values
- a key should be as small as possible (good database design)

Minimality
- no subset of a key can also be designated a key
## Properties of keys

### Adequacy
- uniqueness of key values should be guaranteed by the properties of the application domain
- in other words: it is an error to have different tuples (in the same relation) with the same key values
- a key should be as small as possible (good database design)

### Minimality
- no subset of a key can also be designated a key

### Multiple keys
- there may be more than one key in a relation schema
- one is selected as the **primary** key:
  - cannot be null (**entity integrity**)
  - typically used in **indexing**
Relational model is value-based

There cannot be two different "objects" (here: tuples) whose all attribute values are pairwise equal.

The only way to reference an "object" (tuple) is by providing its key value.

It is not possible to refer to the location of an object (tuple).

These properties are not shared by the ER model, object-oriented models, XML etc.
Relational model is value-based

No duplicates

There cannot be two different “objects” (here: tuples) whose all attribute values are pairwise equal.
Relational model is value-based

**No duplicates**

There cannot be two different “objects” (here: tuples) whose all attribute values are pairwise equal.

**No pointers**

The only way to reference an “object” (tuple) is by providing its key value.
Relational model is value-based

No duplicates
There cannot be two different “objects” (here: tuples) whose all attribute values are pairwise equal.

No pointers
The only way to reference an “object” (tuple) is by providing its key value.

No notion of location
It is not possible to refer to the location of an object (tuple).
Relational model is value-based

No duplicates
There cannot be two different “objects” (here: tuples) whose all attribute values are pairwise equal.

No pointers
The only way to reference an “object” (tuple) is by providing its key value.

No notion of location
It is not possible to refer to the location of an object (tuple).

These properties are not shared by the ER model, object-oriented models, XML etc.
Foreign keys

Relation schemas $R_1$, $R_2$ (not necessarily distinct).
Foreign keys

Relation schemas $R_1$, $R_2$ (not necessarily distinct).

**Foreign key constraint**

A pair of sets of attributes $(S_1, S_2)$ such that:

- $S_1 \subseteq R_1$, $S_2 \subseteq R_2$
- $S_2$ is a key of $R_2$
- the number of attributes and their respective domains in $S_1$ and $S_2$ are the same.
Foreign keys

Relation schemas $R_1, R_2$ (not necessarily distinct).

### Foreign key constraint

A pair of sets of attributes $(S_1, S_2)$ such that:
- $S_1 \subseteq R_1, S_2 \subseteq R_2$
- $S_2$ is a key of $R_2$
- the number of attributes and their respective domains in $S_1$ and $S_2$ are the same.

A pair of instances $(r_1, r_2)$ satisfies a foreign key constraint $(S_1, S_2)$ if for every tuple $t_1 \in r_1$, $t_1[S_1] = t_2[S_2]$ for some tuple $t_2 \in r_2$ or $t_1[S_1]$ is null.
Foreign keys

Relation schemas $R_1, R_2$ (not necessarily distinct).

**Foreign key constraint**

A pair of sets of attributes $(S_1, S_2)$ such that:
- $S_1 \subseteq R_1$, $S_2 \subseteq R_2$
- $S_2$ is a key of $R_2$
- the number of attributes and their respective domains in $S_1$ and $S_2$ are the same.

A pair of instances $(r_1, r_2)$ satisfies a foreign key constraint $(S_1, S_2)$ if for every tuple $t_1 \in r_1$, $t_1[S_1] = t_2[S_2]$ for some tuple $t_2 \in r_2$ or $t_1[S_1]$ is null.

A primary key (or a part thereof) can be a foreign key at the same time (but then it can’t be null).
Other kinds of integrity constraints?

- Functional dependencies generalize key constraints
- Inclusion dependencies generalize foreign key constraints
- Multivalued dependencies are rarely supported by current DBMS
Other kinds of integrity constraints?

**Functional dependencies**
- generalize key constraints

Inclusion dependencies generalize foreign key constraints

Multivalued dependencies are rarely supported by current DBMS.
Other kinds of integrity constraints?

**Functional dependencies**
- generalize key constraints

**Inclusion dependencies**
- generalize foreign key constraints
Other kinds of integrity constraints?

Functional dependencies
- generalize key constraints

Inclusion dependencies
- generalize foreign key constraints

Multivalued dependencies
Other kinds of integrity constraints?

Functional dependencies
- generalize key constraints

Inclusion dependencies
- generalize foreign key constraints

Multivalued dependencies

All rarely supported by current DBMS.
Logical conditions

"I'm willing to admit that I may not always be right, but I am never wrong."
Samuel Goldwyn

Scope can be associated with attributes, tuples, relations, or databases. SQL DBMS often implements only tuple-level conditions (CHECK constraints).
Logical conditions

General conditions

- essentially queries
- shouldn’t evaluate to False in any valid instance

“I’m willing to admit that I may not always be right, but I am never wrong.”

Samuel Goldwyn

Scope can be associated with attributes, tuples, relations, or databases. SQL DBMS often implements only tuple-level conditions (CHECK constraints).
Logical conditions

General conditions

- essentially queries
- shouldn’t evaluate to False in any valid instance

"I’m willing to admit that I may not always be right, but I am never wrong.”

Samuel Goldwyn

Scope

- can be associated with attributes, tuples, relations, or databases
- SQL DBMS often implements only tuple-level conditions (CHECK constraints)
Part II

Relational query languages
Outline of Part II

3. Relational algebra

4. Query evaluation and optimization

5. SQL
Relational query languages

Relational algebra

- a set of algebraic operators
- each operator takes one or two relations as arguments and returns a relation as the result
- operators can be nested to form expressions

Procedural query language: expressions describe how the query can be evaluated

Relational calculus

- a logic language: expressions involve Boolean operators and quantifiers
- declarative query language: expressions do not describe how to evaluate the query
- we will not talk about it

SQL

- a mix of relational algebra and logic (procedural/declarative)
- the standard query language of the existing DBMS.
Relational query languages

Relational algebra

- a set of **algebraic** operators
- each operator takes one or two relations as arguments and returns a relation as the result
- operators can be nested to form **expressions**
- **procedural** query language: expressions describe how the query can be evaluated
Relational query languages

Relational algebra
- a set of algebraic operators
- each operator takes one or two relations as arguments and returns a relation as the result
- operators can be nested to form expressions
- procedural query language: expressions describe how the query can be evaluated

Relational calculus
- a logic language: expressions involve Boolean operators and quantifiers
- declarative query language: expressions do not describe how to evaluate the query
- we will not talk about it

SQL
- a mix of relational algebra and logic (procedural/declarative)
- the standard query language of the existing DBMS.
# Relational query languages

## Relational algebra
- A set of algebraic operators
- Each operator takes one or two relations as arguments and returns a relation as the result
- Operators can be nested to form expressions
- Procedural query language: expressions describe how the query can be evaluated

## Relational calculus
- A logic language: expressions involve Boolean operators and quantifiers
- Declarative query language: expressions do not describe how to evaluate the query
- We will not talk about it

## SQL
- A mix of relational algebra and logic (procedural/declarative)
- The standard query language of the existing DBMS.
Subtle issues

Nulls

relational algebra does not allow nulls

SQL does

Duplicates

relational algebra operates on sets and does not allow duplicates

SQL allows duplicates and operates on multisets (bags)

duplicates irrelevant for most queries

Order

neither relational algebra nor SQL can specify order within sets of tuples

in SQL top-level query results can be ordered

but not in subqueries

Subtle issues

Nulls

- relational algebra does not allow nulls
- SQL does
Subtle issues

**Nulls**
- relational algebra does not allow nulls
- SQL does

**Duplicates**
- relational algebra operates on sets and does not allow duplicates
- SQL allows duplicates and operates on multisets (bags)
- duplicates irrelevant for most queries
# Subtle issues

## Nulls
- relational algebra does not allow nulls
- SQL does

## Duplicates
- relational algebra operates on sets and does not allow duplicates
- SQL allows duplicates and operates on multisets (bags)
- duplicates irrelevant for most queries

## Order
- neither relational algebra nor SQL can specify order within sets of tuples
- in SQL top-level query results can be ordered
  - but not in subqueries
Basic operators

Set operators
- union
- set difference

Relational operators
- Cartesian product
- selection
- projection
- renaming

This is a minimal set of operators.
Basic operators

Set operators

- union
- set difference
Basic operators

Set operators
- union
- set difference

Relational operators
- Cartesian product
- selection
- projection
- renaming.
Basic operators

Set operators
- union
- set difference

Relational operators
- Cartesian product
- selection
- projection
- renaming.

This is a **minimal** set of operators.
Union and difference

Union ($\cup$) of $R_1$ and $R_2$

$\text{arity}(R_1 \cup R_2) = \text{arity}(R_1) = \text{arity}(R_2)$

$t \in r_1 \cup r_2$ iff $t \in r_1$ or $t \in r_2$.

Difference ($-$) of $R_1$ and $R_2$

$\text{arity}(R_1 - R_2) = \text{arity}(R_1) = \text{arity}(R_2)$

$t \in r_1 - r_2$ iff $t \in r_1$ and $t \not\in r_2$.

The arguments of union and difference need to be compatible.

Compatibility of $R_1$ and $R_2$

$\text{arity}(R_1) = \text{arity}(R_2)$ the corresponding attribute domains in $R_1$ and $R_2$ are the same thus compatibility of two relations can be determined solely on the basis of their schemas (compile-time property).
Union and difference

**Union (\(\cup\)) of \(R_1\) and \(R_2\)**

- \(\text{arity}(R_1 \cup R_2) = \text{arity}(R_1) = \text{arity}(R_2)\)
- \(t \in r_1 \cup r_2\) iff \(t \in r_1\) or \(t \in r_2\).
Union and difference

### Union ($\cup$) of $R_1$ and $R_2$

- $arity(R_1 \cup R_2) = arity(R_1) = arity(R_2)$
- $t \in r_1 \cup r_2$ iff $t \in r_1$ or $t \in r_2$.

### Difference ($-$) of $R_1$ and $R_2$

- $arity(R_1 - R_2) = arity(R_1) = arity(R_2)$
- $t \in r_1 - r_2$ iff $t \in r_1$ and $t \not\in r_2$. 

The arguments of union and difference need to be compatible.

Compatibility of $R_1$ and $R_2$

arity($R_1$) = arity($R_2$) the corresponding attribute domains in $R_1$ and $R_2$ are the same thus compatibility of two relations can be determined solely on the basis of their schemas (compile-time property).
Union and difference

Union (\(\cup\)) of \(R_1\) and \(R_2\)
- \(arity(R_1 \cup R_2) = arity(R_1) = arity(R_2)\)
- \(t \in r_1 \cup r_2 \iff t \in r_1 \text{ or } t \in r_2\).

Difference (\(-\)) of \(R_1\) and \(R_2\)
- \(arity(R_1 - R_2) = arity(R_1) = arity(R_2)\)
- \(t \in r_1 - r_2 \iff t \in r_1 \text{ and } t \notin r_2\).

The arguments of union and difference need to be compatible.
Union and difference

**Union (∪) of** $R_1$ and $R_2$

- $arity(R_1 \cup R_2) = arity(R_1) = arity(R_2)$
- $t \in r_1 \cup r_2$ iff $t \in r_1$ or $t \in r_2$.

**Difference (−) of** $R_1$ and $R_2$

- $arity(R_1 - R_2) = arity(R_1) = arity(R_2)$
- $t \in r_1 - r_2$ iff $t \in r_1$ and $t \notin r_2$.

The arguments of union and difference need to be compatible.

**Compatibility of** $R_1$ and $R_2$

- $arity(R_1) = arity(R_2)$
- the corresponding attribute domains in $R_1$ and $R_2$ are the same
- thus compatibility of two relations can be determined solely on the basis of their schemas (compile-time property).
Cartesian product of $R_1$ and $R_2$

$arity(R_1) = k_1, \ arity(R_2) = k_2$

**Cartesian product ($\times$)**

- $arity(R_1 \times R_2) = arity(R_1) + arity(R_2)$
- $t \in r_1 \times r_2$ iff:
  - the first $k_1$ components of $t$ form a tuple in $r_1$, and
  - the next $k_2$ components of $t$ form a tuple in $r_2$. 
Selection condition $E$ built from:
- comparisons between operands which can be constants or attribute names
- Boolean operators: $\land$ (AND), $\lor$ (OR), $\neg$ (NOT).

Selection $\sigma_E(R)$
- $\text{arity}(\sigma_E(R)) = \text{arity}(R)$
- $t \in \sigma_E(r)$ iff $t \in r$ and $t$ satisfies $E$. 
Projection

\[ A_1, \ldots, A_k: \text{ distinct attributes of } R. \]

Projection \( \pi_{A_1, \ldots, A_k}(R) \)

- \( \text{arity}(\pi_{A_1, \ldots, A_k}(R)) = k \)
- \( t \in \pi_{A_1, \ldots, A_k}(r) \) iff for some \( s \in r, \) \( t[A_1] = s[A_1], \ldots, t[A_k] = s[A_k]. \)
Renaming

\[ A_1, \ldots, A_n : \text{attributes of } R \]
\[ B_1, \ldots, B_n : \text{new attributes} \]

Renaming \( R(B_1, \ldots, B_n) \)

- \( \text{arity}(R(B_1, \ldots, B_n)) = \text{arity}(R) = n, \)
- \( t \in r(B_1, \ldots, B_n) \) iff for some \( s \in r, \) \( t[B_1] = s[A_1], \ldots, t[B_n] = s[A_n]. \)
Derived operators

1. Intersection.
2. Quotient.
3. $\theta$-join.
4. Natural join.
Intersection

- \( \text{arity}(R_1 \cap R_2) = \text{arity}(R_1) = \text{arity}(R_2) \)
- \( t \in r_1 \cap r_2 \) iff \( t \in r_1 \) and \( t \in r_2 \).
Intersection

- $arity(R_1 \cap R_2) = arity(R_1) = arity(R_2)$
- $t \in r_1 \cap r_2$ iff $t \in r_1$ and $t \in r_2$.

Intersection is a derived operator:

$$R_1 \cap R_2 = R_1 - (R_1 - R_2).$$
**Quotient**

\[ A_1, \ldots, A_{n+k} : \text{all the attributes of } R_1 \]
\[ A_{n+1}, \ldots, A_{n+k} : \text{all the attributes of } R_2 \]
\[ r_2 \text{ nonempty.} \]

**Quotient (division)**

- \( \text{arity}(R_1 \div R_2) = \text{arity}(R_1) - \text{arity}(R_2) = n \)
- \( t \in r_1 \div r_2 \) iff for all \( s \in r_2 \) there is a \( w \in r_1 \) such that
  - \( t[A_1] = w[A_1], \ldots, t[A_n] = w[A_n], \) and
  - \( s[A_{n+1}] = w[A_{n+1}], \ldots, s[A_{n+k}] = w[A_{n+k}] \).
Quotient

\( A_1, \ldots, A_{n+k} \): all the attributes of \( R_1 \)
\( A_{n+1}, \ldots, A_{n+k} \): all the attributes of \( R_2 \)
\( r_2 \) nonempty.

Quotient (division)

- \( \text{arity}(R_1 \div R_2) = \text{arity}(R_1) - \text{arity}(R_2) = n \)
- \( t \in r_1 \div r_2 \) iff for all \( s \in r_2 \) there is a \( w \in r_1 \) such that
  - \( t[A_1] = w[A_1], \ldots, t[A_n] = w[A_n] \), and
  - \( s[A_{n+1}] = w[A_{n+1}], \ldots, s[A_{n+k}] = w[A_{n+k}] \).

Quotient is a derived operator:

\[
R_1 \div R_2 = \pi_{A_1, \ldots, A_n}(R_1) - \\
\pi_{A_1, \ldots, A_n}(\pi_{A_1, \ldots, A_n}(R_1) \times R_2) - R_1
\]
\(\theta\)-join

\(\theta\): a comparison operator \((=,\neq, <, >, \geq, \leq)\)

\(A_1, \ldots, A_n\): all the attributes of \(R_1\)

\(B_1, \ldots, B_k\): all the attributes of \(R_2\)

**\(\theta\)-join**

\[
\text{arity}(R_1 \bowtie_{A_i \theta B_j} R_2) = \text{arity}(R_1) + \text{arity}(R_2)
\]

\[
R_1 \bowtie_{A_i \theta B_j} R_2 = \sigma_{A_i \theta B_j}(R_1 \times R_2)
\]
### $\theta$-join

$\theta$: a comparison operator ($=, \neq, <, >, \geq, \leq$)

$A_1, \ldots, A_n$: all the attributes of $R_1$

$B_1, \ldots, B_k$: all the attributes of $R_2$

### $\theta$-join

- $\text{arity}(R_1 \bowtie_{A_i \theta B_j} R_2) = \text{arity}(R_1) + \text{arity}(R_2)$

- $R_1 \bowtie_{A_i \theta B_j} R_2 = \sigma_{A_i \theta B_j}(R_1 \times R_2)$

---

### Equijoin

$\theta$-join where $\theta$ is equality.
Natural join

\[ A_1, \ldots, A_n: \text{all the attributes of } R_1 \]
\[ B_1, \ldots, B_k: \text{all the attributes of } R_2 \]
\[ m - \text{the number of attributes common to } R_1 \text{ and } R_2 \]

\[ \text{Natural join arity}(R_1 \bowtie R_2) = \text{arity}(R_1) + \text{arity}(R_2) - m \]
\[ \text{to obtain } r_1 \bowtie r_2: \]
1. select from \( r_1 \times r_2 \) the tuples that agree on all attributes common to \( R_1 \) and \( R_2 \)
2. project duplicate columns out from the resulting tuples.
Query evaluation

Basic
- queries evaluated bottom-up: an operator is applied after the arguments have been computed
- temporary relations for intermediate results

Advanced
- using indexes, sorting and hashing
- special algorithms
- input/output streams, blocking
- parallelism
Query evaluation

Basic

- queries evaluated **bottom-up**: an operator is applied after the arguments have been computed
- temporary relations for intermediate results
## Query evaluation

### Basic
- queries evaluated **bottom-up**: an operator is applied after the arguments have been computed
- temporary relations for intermediate results

### Advanced
- using indexes, sorting and hashing
- special algorithms
- input/output streams, blocking
- parallelism
Indexing

Fast access to individual rows using the values of one or more index columns.

Used to implement:
- selection: atomic conditions ($\sigma_{A=c}$), conjunctive conditions
- equijoin
Indexing

Fast access to individual rows using the values of one or more index columns.

Used to implement:
- selection: atomic conditions \((\sigma_{A=c})\), conjunctive conditions
- equijoin

Underlying technologies:
- B-trees
- hashing
- ...
Query optimization
Query optimization

Logical query optimization
- algebraic laws
- rewrite rules
Query optimization

**Logical query optimization**
- algebraic laws
- rewrite rules

**Cost-based query optimization**
- cost analysis of evaluation plans
- enumeration of evaluation plans
Algebraic laws (examples)

Join reordering

\[
E_1 \bowtie E_2 = E_2 \bowtie E_1 = (E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3).
\]

Pushing selection

\[
F(E_1 \times E_2) = F(E_1) \times E_2 \quad \text{if } F \text{ involves only the attributes of } E_1.
\]

\[
F(E_1 \cup E_2) = F(E_1) \cup F(E_2).
\]

\[
F(E_1 - E_2) = F(E_1) - F(E_2).
\]
Algebraic laws (examples)

Join reordering

\[ E_1 \bowtie E_2 = E_2 \bowtie E_1 \]

\[ (E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3). \]
Algebraic laws (examples)

### Join reordering

\[ E_1 \bowtie E_2 = E_2 \bowtie E_1 \]

\[(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3).\]

### Pushing selection

\[ \sigma_F(E_1 \times E_2) = \sigma_F(E_1) \times E_2 \quad \text{if } F \text{ involves only the attributes of } E_1 \]

\[ \sigma_F(E_1 \cup E_2) = \sigma_F(E_1) \cup \sigma_F(E_2) \]

\[ \sigma_F(E_1 - E_2) = \sigma_F(E_1) - \sigma_F(E_2) \]
SQL

Support
- virtually all relational DBMS
- vendor-specific extensions

Standardized (partially)
- SQL2 or SQL-92 (completed 1992)
- SQL3, SQL:1999, SQL:2003 (completed)
- SQL:2006 (ongoing work)
SQL language components

- query language
- data definition language
- data manipulation language
- integrity constraints and views
- API’s (ODBC, JDBC)
- host language preprocessors (Embedded SQL, SQLJ)
- support XML data and queries
- ...

Relational databases
Basic SQL queries

**Basic form**

```sql
SELECT A_1, \ldots, A_n
FROM R_1, \ldots, R_k
WHERE C
```

**Corresponding relational algebra expression**

\[
\pi_{A_1, \ldots, A_n}(\sigma_C(R_1 \times \cdots \times R_k))
\]
Range variables

To refer to a relation more than once in the FROM clause, range variables are used.

Example

```sql
SELECT R1.A, R2.B
FROM R R1,R R2
WHERE R1.B=R2.A
```

corresponds to

\[
\pi_{A,D}(R(A, B) \bowtie_{B=C} R(C, D)).
\]
Manipulating the result

SELECT *: all the columns are selected.

SELECT DISTINCT: duplicates are eliminated from the result.

ORDER BY $A_1, \ldots, A_m$: the result is sorted according to $A_1, \ldots, A_m$.

$E$ AS $A$ can be used instead of an column $A$ in the SELECT list to mean that the value of the column $A$ in the result is determined using the (arithmetic or string) expression $E$. 
Set operations

**UNION** set union.

**INTERSECT** set intersection.

**EXCEPT** set difference.

Note

- INTERSECT and EXCEPT can be expressed using other SQL constructs
Nested queries

A query $Q$ can appear as a subquery in the $\textit{WHERE}$ clause which can now contain:

- $A \in Q$: for set membership ($A \in Q$)
- $A \not\in Q$: for the negation of set membership ($A \not\in Q$)
- $A \textit{ALL} Q$: $A$ is in the relationship to all the elements of $Q$ ($\in \{=, <, >\}$)
- $A \textit{ANY} Q$: $A$ is in the relationship to some elements of $Q$
- $\exists Q$: $Q$ is nonempty
- $\not\exists Q$: $Q$ is empty

Notes

The subqueries can contain columns from enclosing queries. Multiple occurrences of the same column name are disambiguated by choosing the closest enclosing $\textit{FROM}$ clause.
Nested queries

**Subquery**

A query $Q$ can appear as a subquery in the WHERE clause which can now contain:

- **A IN Q**: for set membership ($A \in Q$)
- **A NOT IN Q**: for the negation of set membership ($A \notin Q$)
- **A $\theta$ ALL Q**: $A$ is in the relationship $\theta$ to all the elements of $Q$
  ($\theta \in \{=, <, >, \geq, \leq, <>\}$)
- **A $\theta$ ANY Q**: $A$ is in the relationship $\theta$ to some elements of $Q$
- **EXISTS Q**: $Q$ is nonempty
- **NOT EXISTS Q**: $Q$ is empty

Notes

The subqueries can contain columns from enclosing queries

Multiple occurrences of the same column name are disambiguated by choosing the closest enclosing FROM clause.
Nested queries

Subquery

A query $Q$ can appear as a subquery in the WHERE clause which can now contain:

- $A \text{ IN } Q$: for set membership ($A \in Q$)
- $A \text{ NOT IN } Q$: for the negation of set membership ($A \not\in Q$)
- $A \theta \text{ ALL } Q$: $A$ is in the relationship $\theta$ to all the elements of $Q$ ($\theta \in \{=, <, >, \geq, \leq, <>\}$)
- $A \theta \text{ ANY } Q$: $A$ is in the relationship $\theta$ to some elements of $Q$
- $\text{EXISTS } Q$: $Q$ is nonempty
- $\text{NOT EXISTS } Q$: $Q$ is empty

Notes

- the subqueries can contain columns from enclosing queries
- multiple occurrences of the same column name are disambiguated by choosing the closest enclosing FROM clause.
Aggregation

Instead of a column $A$, the SELECT list can contain the results of some aggregate function applied to all the values in the column $A$ in the relation.

Aggregation functions:

- $\text{COUNT}(A)$: the number of all values in the column $A$ (with duplicates)
- $\text{SUM}(A)$: the sum of all values in the column $A$ (with duplicates)
- $\text{AVG}(A)$: the average of all values in the column $A$ (with duplicates)
- $\text{MAX}(A)$: the maximum value in the column $A$
- $\text{MIN}(A)$: the minimum value in the column $A$.

Notes:

$\text{DISTINCT} A$, instead of $A$, considers only distinct values.

Aggregation queries not expressible in relational algebra.
Instead of a column $A$, the SELECT list can contain the results of some aggregate function applied to all the values in the column $A$ in the relation.
Aggregation

Instead of a column $A$, the SELECT list can contain the results of some aggregate function applied to all the values in the column $A$ in the relation.

Aggregation functions

- $\text{COUNT}(A)$: the number of all values in the column $A$ (with duplicates)
- $\text{SUM}(A)$: the sum of all values in the column $A$ (with duplicates)
- $\text{AVG}(A)$: the average of all values in the column $A$ (with duplicates)
- $\text{MAX}(A)$: the maximum value in the column $A$
- $\text{MIN}(A)$: the minimum value in the column $A$.
Aggregation

Instead of a column $A$, the SELECT list can contain the results of some aggregate function applied to all the values in the column $A$ in the relation.

**Aggregation functions**

- $\text{COUNT}(A)$: the number of all values in the column $A$ (with duplicates)
- $\text{SUM}(A)$: the sum of all values in the column $A$ (with duplicates)
- $\text{AVG}(A)$: the average of all values in the column $A$ (with duplicates)
- $\text{MAX}(A)$: the maximum value in the column $A$
- $\text{MIN}(A)$: the minimum value in the column $A$.

**Notes**

- $\text{DISTINCT} A$, instead of $A$, considers only distinct values
- aggregation queries not expressible in relational algebra
Grouping

The clause `GROUP BY` assembles the tuples in the result of the query into groups with identical values in columns $A_1, \ldots, A_n$.

The clause `HAVING` leaves only those groups that satisfy the condition $C$.

Notes

The `SELECT` list of a query with `GROUP BY` can contain only:
- the columns mentioned in `GROUP BY` (or expressions with those), or
- the result of an aggregate function, which is then viewed as applied group-by-group.
Grouping

The clause

GROUP BY $A_1, \ldots, A_n$

assembles the tuples in the result of the query into groups with identical values in columns $A_1, \ldots, A_n$. 
Grouping

The clause

**GROUP BY** $A_1, \ldots, A_n$

assembles the tuples in the result of the query into groups with identical values in columns $A_1, \ldots, A_n$.

The clause

**HAVING** $C$

leaves only those groups that satisfy the condition $C$. 
Grouping

The clause

\texttt{GROUP BY } A_1, \ldots, A_n

assembles the tuples in the result of the query into groups with identical values in columns \( A_1, \ldots, A_n \).

The clause

\texttt{HAVING } C

leaves only those groups that satisfy the condition \( C \).

Notes

The \texttt{SELECT} list of a query with \texttt{GROUP BY} can contain only:

- the columns mentioned in \texttt{GROUP BY} (or expressions with those), or
- the result of an aggregate function, which is then viewed as applied group-by-group.
Building complex queries

A complex query can be broken up into smaller pieces using:

- nested queries in the `FROM` clause
- views

**View**

Computed relation whose contents are defined by an SQL query.

**Creating a view**

```
CREATE VIEW View-name (
Attr1,...,
Attrn
) AS Query
```

**Dropping a view**

```
DROP VIEW View-name
```
Building complex queries

A complex query can be broken up into smaller pieces using:
- nested queries in the FROM clause
- views.
Building complex queries

A complex query can be broken up into smaller pieces using:
- nested queries in the FROM clause
- views.

View

Computed relation whose contents are defined by an SQL query.
Building complex queries

A complex query can be broken up into smaller pieces using:
- nested queries in the FROM clause
- views.

View
Computed relation whose contents are defined by an SQL query.

Creating a view

CREATE VIEW View-name(Attr1, ..., Attrn) AS Query
Building complex queries

A complex query can be broken up into smaller pieces using:
- nested queries in the FROM clause
- views.

**View**

Computed relation whose contents are defined by an SQL query.

**Creating a view**

CREATE VIEW View-name(Attr1,...,Attrn) AS Query

**Dropping a view**

DROP VIEW View-name
Nulls

Various interpretations: unknown, missing value, inapplicable, no information...

In SQL columns that are not explicitly or implicitly designated as NOT NULL can contain nulls.

Behavior of nulls comparisons return the unknown truth value if at least one of the arguments is null. IS NULL returns true.

Null values counted by COUNT(*), discarded by other aggregate operators.
Nulls

Various interpretations: *unknown*, missing value, inapplicable, no information...
Various interpretations: *unknown*, missing value, inapplicable, no information...

In SQL columns that are not explicitly or implicitly designated as \texttt{NOT NULL} can contain \texttt{nulls}.
Nulls

Various interpretations: unknown, missing value, inapplicable, no information...

In SQL columns that are not explicitly or implicitly designated as NOT NULL can contain nulls.

Behavior of nulls

- comparisons return the unknown truth value if at least one of the arguments is null
- IS NULL returns true
- null values counted by COUNT(*), discarded by other aggregate operators.
Three-valued logic

<table>
<thead>
<tr>
<th>NOT</th>
<th>T</th>
<th>F</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AND</th>
<th>T</th>
<th>F</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>?</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>F</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>T</th>
<th>F</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>?</td>
</tr>
<tr>
<td>?</td>
<td>T</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Outer joins

To keep the tuples in the result if there are no matching tuples in the other argument of the join:

- **LEFT**: preserve only the tuples from the left argument
- **RIGHT**: preserve only the tuples from the right argument
- **FULL**: preserve the tuples from both arguments.

The result tuples are padded with nulls.

Syntax (in the FROM clause):

```
R1 OUTER JOIN R2 ON Condition USING Columns
```

Notes

Outer joins can be expressed using other SQL constructs. Some DBMS, e.g., Oracle, use a different syntax for outer joins.
Outer joins

To keep the tuples in the result if there are no matching tuples in the other argument of the join:

- **LEFT**: preserve only the tuples from the left argument
- **RIGHT**: preserve only the tuples from the right argument
- **FULL**: preserve the tuples from both arguments.

The result tuples are padded with nulls.
Outer joins

To keep the tuples in the result if there are no matching tuples in the other argument of the join:

- **LEFT**: preserve only the tuples from the left argument
- **RIGHT**: preserve only the tuples from the right argument
- **FULL**: preserve the tuples from both arguments.

The result tuples are padded with nulls.

Syntax (in the FROM clause):

\[ R_1 \text{ OUTER JOIN } R_2 \text{ ON } \text{Condition} \text{ USING } \text{Columns} \]
Outer joins

To keep the tuples in the result if there are no matching tuples in the other argument of the join:

- **LEFT**: preserve only the tuples from the left argument
- **RIGHT**: preserve only the tuples from the right argument
- **FULL**: preserve the tuples from both arguments.

The result tuples are padded with nulls.

Syntax (in the FROM clause):

\[ R_1 \text{ OUTER JOIN } R_2 \text{ ON } \text{Condition USING Columns} \]

Notes

- outer joins can be expressed using other SQL constructs
- some DBMS, e.g., Oracle, use a different syntax for outer joins.
Limitations of relational query languages

They cannot express queries involving transitive closure of binary relations:

"List all the ancestors of David."

"Find all the buildings reachable from Bell Hall without going outside."

Solution

Recursive views.
Limitations of relational query languages

They cannot express queries involving transitive closure of binary relations:

- “List all the ancestors of David.”
- “Find all the buildings reachable from Bell Hall without going outside.”
Limitations of relational query languages

They cannot express queries involving transitive closure of binary relations:

- “List all the ancestors of David.”
- “Find all the buildings reachable from Bell Hall without going outside.”

Solution

Recursive views.
Recursion in SQL3

A relation \( R \) depends on a relation \( S \) if \( S \) is used, directly or indirectly, in the definition of \( R \).

In a recursive view definition a relation may depend on itself!

Recursive views in SQL3, still unsupported in most DBMS recursively defined relations should be preceded by `RECURSIVE`.

Syntax:

```
WITH R AS definition of R
query to R
```
Recursion in SQL3

A relation $R$ depends on a relation $S$ if $S$ is used, directly or indirectly, in the definition of $R$. 

SQL3, still unsupported in most DBMS recursively defined relations should be preceded by `RECURSIVE` syntax: `WITH R AS definition of R query to R`
Recursion in SQL3

A relation \( R \) \textbf{depends} on a relation \( S \) if \( S \) is used, directly or indirectly, in the definition of \( R \).

In a recursive view definition a relation may depend on itself!
Recursion in SQL3

A relation $R$ depends on a relation $S$ if $S$ is used, directly or indirectly, in the definition of $R$.

In a recursive view definition a relation may depend on itself!

Recursive views in SQL

- SQL3, still unsupported in most DBMS
- recursively defined relations should be preceded by RECURSIVE.
- syntax:
  
  ```sql
  WITH R AS
  definition of R
  query to R
  ```
Find all the ancestors of David:

```sql
WITH RECURSIVE Anc(Upper, Lower) AS
  (SELECT * FROM Parent)
UNION
  (SELECT P.Upper, A.Lower
   FROM Parent AS P, Anc AS A
   WHERE P.Lower=A.Upper)

SELECT Anc.Upper
FROM Anc
WHERE Anc.Lower='David';
```

Stratification restriction

No view can depend on itself through `EXCEPT` or aggregation.
Example

Find all the ancestors of David:

```sql
WITH RECURSIVE Anc(Upper,Lower) AS
  (SELECT * FROM Parent)
  UNION
  (SELECT P.Upper, A.Lower FROM Parent AS P, Anc AS A
   WHERE P.Lower=A.Upper)

SELECT Anc.Upper
FROM Anc
WHERE Anc.Lower='David';
```
Example

Find all the ancestors of David:

WITH RECURSIVE Anc(Upper,Lower) AS
  (SELECT * FROM Parent)
 UNION
  (SELECT P.Upper, A.Lower
   FROM Parent AS P, Anc AS A
   WHERE P.Lower=A.Upper)

SELECT Anc.Upper
FROM Anc
WHERE Anc.Lower='David';

Stratification restriction

No view can depend on itself through EXCEPT or aggregation.
Evaluating queries with recursive views

Evaluation algorithm
1. Initially, the contents of all views are empty.
2. Compute the new contents of the views, using database relations and the current contents of the views.
3. Repeat the previous step until no changes in view contents occur.

Why does this terminate?
Evaluating queries with recursive views

Evaluation algorithm

1. Initially, the contents of all views are empty.
2. Compute the new contents of the views, using database relations and the current contents of the views.
3. Repeat the previous step until no changes in view contents occur.
Evaluating queries with recursive views

Evaluation algorithm

1. Initially, the contents of all views are empty.
2. Compute the new contents of the views, using database relations and the current contents of the views.
3. Repeat the previous step until no changes in view contents occur.

Why does this terminate?