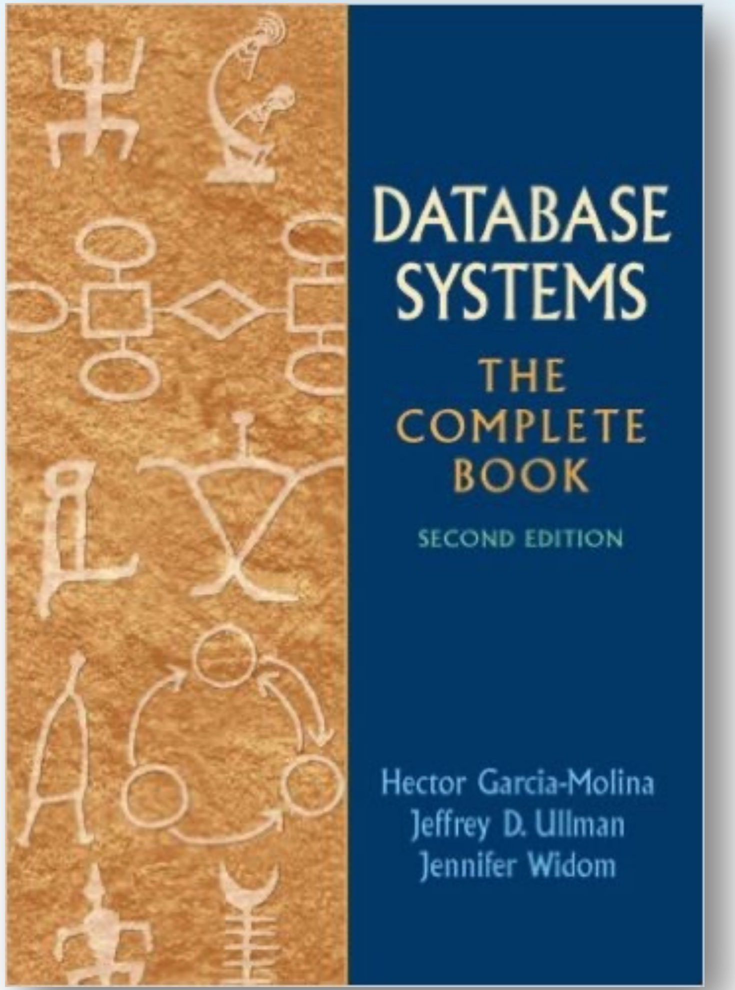
DAS 839 NoSQL Systems

Structured Query Language (SQL)

Vinu E. Venugopal, IIIT Bangalore

Background Literature



Database Systems – The Complete Book (2nd Ed)
Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom. Pearson Prentice Hall 2009.
ISBN: 978-0131873254

- Chapter 6: The Database Language SQL
- Chapter 7: Constraints & Triggers
- Chapter 8: Views & Indexes
- Chapter 9: SQL in a Server Environment

1. Basic SQL Syntax

We will generally follow the **ANSI SQL:99** standardized syntax for SQL. This is supported by most current DBMS vendors, including Microsoft SQL Server, MySQL, PostgreSQL, Oracle (from version 9i on, Oracle is mostly compliant to SQL:99), and many more…

Still, SQL:99 is not yet fully supported by all of the big players: IBM's DB2, for example, still (mostly) follows SQL:92.

Oracle 11g SQL Manual:
https://docs.oracle.com/cd/E11882_01/server.112/e41084.pdf

Differences between ANSI SQL:99 and Oracle:
http://www-db.stanford.edu/~ullman/fcdb/oracle/or-nonstandard.html

Philip Greenspun's blog "SQL for Web Nerds":
http://philip.greenspun.com/sql/introduction.html

A Note on the Various SQL Standards

The core of SQL has not changed much since 1992.
See, e.g.: http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt, for the original specification (~700 A4 pages!)

SQL:2011 (the latest revision), for example, is still backward-compatible to SQL:92 and additionally introduces temporal operators and versioning for tuples.
See also: "What's New in SQL:2011"

SQL adopts the **Relational Model** and provides very detailed **declarative language-constructs** that allow users to specify and manipulate relation schemas, database schemas, and relation instances.

Most important difference: SQL deviates from its theoretical foundation, **Relational Algebra**, in one important way:

- Relation instances are **finite bags of tuples** rather than finite sets.
- If ordering is considered, relations are even **finite lists of tuples**.

A Note on the Various SQL Standards

The core of SQL has not changed much since 1992.
See, e.g.: http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt, for the original specification (~700 A4 pages!)

SQL:2011 (the latest revision), for example, is still backward-compatible to SQL:92 and additionally introduces temporal operators and versioning for tuples.
See also: "What's New in SQL:2011"

SQL adopts the **Relational Model** and provides very detailed **declarative language-constructs** that allow users to specify and manipulate relation schemas, database schemas, and relation instances.

Most important difference: SQL deviates from its theoretical foundation, **Relational Algebra**, in one important way:

- Relation instances are **finite bags of tuples** rather than finite sets.
- If ordering is considered, relations are even **finite lists of tuples**.

A Note on the Various SQL Standards

The core of SQL has not changed much since 1992.
See, e.g.: http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt, for the original specification (~700 A4 pages!)

SQL:2011 (the latest revision), for example, is still backward-compatible to SQL:92 and additionally introduces temporal operators and versioning for tuples.
See also: "What's New in SQL:2011"

SQL adopts the **Relational Model** and provides very detailed **declarative language-constructs** that allow users to specify and manipulate relation schemas, database schemas, and relation instances.

Most important difference: SQL deviates from its theoretical foundation, **Relational Algebra**, in one important way:

- Relation instances are **finite bags of tuples** rather than finite sets.
- If ordering is considered, relations are even **finite lists of tuples**.

Overview of Relational Operations

Operands

In SQL, all **operands** are relations, denoted as either *R* (schema-level) or *r* (instance-level).

Operations

- **Set-Operations**: $\cup$, $\cap$, $-$ (following the common semantics of set operations)
- **Selection**: $\sigma_C(R)$ (selects those tuples from *R* for which a Boolean condition *C* evaluates to *true*)
- **Projection**: $\pi_{A1,\ldots,As}(R)$ (removes attributes from *R* that are not in $A_1,\ldots,A_s$ and eliminates resulting duplicates)
- **Renaming**: $\rho_\gamma(R)$ (renames the attributes of a relation using mapping γ)
- **Cross-Product**: $R \times S$ (combines all tuples from *R* with all tuples in *S*)
- **Natural-Join**: $R \bowtie S$ (joins two relations based on their common attributes)
- **Condition-Join**: $R \bowtie_C S$ (joins two relations based on a Boolean condition *C*)
- **Semi-Join**: $R \ltimes S$ (like natural-join, but keeps only attributes from *R*)
- **Outer-Joins**: $R ⟗ S$ (augments tuples without join partner with NULL's)
- **Grouping/Aggregations**: $\alpha_{A,COUNT(B)\mapsto N}(R)$ (groups and aggregates tuples; built-in aggregations include MIN, MAX, COUNT, SUM, AVG)

Relational Database Engines & Big Market Players

ORACLE®

MySQL®

PostgreSQL

monetdb

IBM® DB2®

Microsoft SQL Server

SYBASE

SAP HANA

Sample Database System: PostgreSQL

PostgreSQL ("Postgres" for short) is an open-source database system which has originally been developed as a research prototype at MIT.
PostgreSQL itself is based on Ingres (UC Berkeley, 1970's until today):

http://www.postgresql.org/

All of the following SQL examples can directly be tried out in PostgreSQL!

SQL Sublanguages

SQL consists of several submodules (also called "sublanguages"):

- **DDL** (Data Definition Language)

  CREATE / DROP TABLE, CREATE / DROP INDEX, CREATE OR REPLACE VIEW, etc.

- **DML** (Data Manipulation Language)

  INSERT, UPDATE, DELETE, etc.

- **DQL** (Data Query Language) *

  SELECT … FROM … WHERE

- **DCL** (Data Control Language)

  GRANT, REVOKE, etc.

- **TCL** (Transaction Control Language)

  BEGIN, ABORT, COMMIT, etc.

* DML and DQL are usually combined to just "DML".

SQL Sublanguages

SQL consists of several submodules (also called "sublanguages"):

- **DDL** (Data Definition Language)

  CREATE / DROP TABLE, CREATE / DROP INDEX, CREATE OR REPLACE VIEW, etc.

- **DML** (Data Manipulation Language)

  INSERT, UPDATE, DELETE, etc.

- **DQL** (Data Query Language) *

  SELECT … FROM … WHERE

- **DCL** (Data Control Language)

  GRANT, REVOKE, etc.

- **TCL** (Transaction Control Language)

  BEGIN, ABORT, COMMIT, etc.

* DML and DQL are usually combined to just "DML".

SQL Sublanguages

SQL consists of several submodules (also called "sublanguages"):

- **DDL** (Data Definition Language)

  CREATE / DROP TABLE, CREATE / DROP INDEX, CREATE OR REPLACE VIEW, etc.

- **DML** (Data Manipulation Language)

  INSERT, UPDATE, DELETE, etc.

- **DQL** (Data Query Language) *

  SELECT … FROM … WHERE

- **DCL** (Data Control Language)

  GRANT, REVOKE, etc.

- **TCL** (Transaction Control Language)

  BEGIN, ABORT, COMMIT, etc.

Begin

* DML and DQL are usually combined to just "DML".

SQL Sublanguages

SQL consists of several submodules (also called "sublanguages"):

- **DDL** (Data Definition Language)

  CREATE / DROP TABLE, CREATE / DROP INDEX, CREATE OR REPLACE VIEW, etc.

- **DML** (Data Manipulation Language)

  INSERT, UPDATE, DELETE, etc.

- **DQL** (Data Query Language) *

  SELECT … FROM … WHERE

- **DCL** (Data Control Language)

  GRANT, REVOKE, etc.

- **TCL** (Transaction Control Language)

  BEGIN, ABORT, COMMIT, etc.

Begin

Commit

* DML and DQL are usually combined to just "DML".

Data Definition Language (DDL)

We write a **relation schema** as follows:
Movie(title, year, length, inColor, studioName, producerCertN)

Here, Movie is the **relation name** and title, year, length are its **attributes.**

This relation schema of Movie (incl. its attributes and their data types) can be defined via a CREATE TABLE statement in SQL as follows:

```
CREATE TABLE Movie (
  title       VARCHAR(255),
  year        INTEGER,
  length      INTEGER,
  inColor     CHAR(1),
  studioName  VARCHAR(255),
  producerCertN INTEGER);
```

Data Definition Language (DDL)

We write a **relation schema** as follows:
Movie(title, year, length, inColor, studioName, producerCertN)

Here, Movie is the **relation name** and title, year, length are its **attributes.**

This relation schema of Movie (incl. its attributes and their data types) can be defined via a CREATE TABLE statement in SQL as follows:

```
CREATE TABLE Movie (
  title        VARCHAR(255),
  year         INTEGER,
  length       INTEGER,
  inColor      CHAR(1),
  studioName   VARCHAR(255),
  producerCertN INTEGER);
```

Data Definition Language (DDL)

We write a **relation schema** as follows:
Movie(title, year, length, inColor, studioName, producerCertN)

Here, Movie is the **relation name** and title, year, length are its **attributes.**

This relation schema of Movie (incl. its attributes and their data types) can be defined via a CREATE TABLE statement in SQL as follows:

```
CREATE TABLE Movie (
  title       VARCHAR(255),
  year        INTEGER,
  length      INTEGER,
  inColor     CHAR(1),
  studioName  VARCHAR(255),
  producerCertN  INTEGER);
```

Data Definition Language (DDL)

We write a **relation schema** as follows:
Movie(title, year, length, inColor, studioName, producerCertN)

Here, Movie is the **relation name** and title, year, length are its **attributes.**

This relation schema of Movie (incl. its attributes and their data types) can be defined via a CREATE TABLE statement in SQL as follows:

```
CREATE TABLE Movie (
  title        VARCHAR(255),
  year         INTEGER,
  length       INTEGER,
  inColor      CHAR(1),
  studioName   VARCHAR(255),
  producerCertN  INTEGER);
```

A **database schema** is the set of all relation schemas in the database. Most DBMS's allow multiple database schemas to be created and maintained aside each other.

The relation schema can be "dropped" (i.e., be removed) from the database schema as follows: **DROP TABLE** [IF EXISTS] Movie;

Overview of SQL Data Types

Common SQL data types (defined by the ANSI standard):

- BOOLEAN, BINARY(*n*) : binary types with up to *n* bits
- CHAR(*n*) : fixed-length character string of *n* characters
- VARCHAR(*n*) : variable-length character string of up to *n* characters
- INTEGER / INT, BIGINT, SMALLINT, FLOAT(*p*), NUMERIC(*n*,*d*), DECIMAL(*n*,*d*), REAL, DOUBLE PRECISION : numerical data types with a fixed or variable precision either using *p* bits (for the mantissa) or using *n* as the total number of digits, with *d* digits to the right of the dot
- DATE / TIME : values have the form DATE '<*m*><*m*>/<*d*><*d*>/<*y*><*y*><*y*><*y*>' for specifying a month, day, year, etc. (usually with vendor specific conversion conventions)
- Other variable-length "large" data types: TEXT, BLOB, CLOB, XML, etc.

SQL has type-specific conversions and arithmetic operations:

```
SELECT name
FROM movieStar
WHERE birthdate > DATE '09/28/1980';
```

See, for example, http://www.w3schools.com/sqL/sql_datatypes_general.asp for a detailed description of SQL data types.

Overview of SQL Data Types

Common SQL data types (defined by the ANSI standard):

- BOOLEAN, BINARY(*n*) : binary types with up to *n* bits
- CHAR(*n*) : fixed-length character string of *n* characters
- VARCHAR(*n*) : variable-length character string of up to *n* characters
- INTEGER / INT, BIGINT, SMALLINT, FLOAT(*p*), NUMERIC(*n*,*d*), DECIMAL(*n*,*d*), REAL, DOUBLE PRECISION : numerical data types with a fixed or variable precision either using *p* bits (for the mantissa) or using *n* as the total number of digits, with *d* digits to the right of the dot
- DATE / TIME : values have the form DATE '<m><m>/<d><d>/<y><y><y><y>' for specifying a month, day, year, etc. (usually with vendor specific conversion conventions)
- Other variable-length "large" data types: TEXT, BLOB, CLOB, XML, etc.

SQL has type-specific conversions and arithmetic operations:

```
SELECT name
FROM movieStar
WHERE birthdate > DATE '09/28/1980';
```

See, for example, http://www.w3schools.com/sqL/sql_datatypes_general.asp for a detailed description of SQL data types.

Overview of SQL Data Types

Common SQL data types (defined by the ANSI standard):

CHAR(10) [NOSQLXXXXX]

VARCHAR(10) [NOSQL'\0']

- BOOLEAN, BINARY(*n*) : binary types with up to *n* bits
- CHAR(*n*) : fixed-length character string of *n* characters
- VARCHAR(*n*) : variable-length character string of up to *n* characters
- INTEGER / INT, BIGINT, SMALLINT, FLOAT(*p*), NUMERIC(*n*,*d*), DECIMAL(*n*,*d*), REAL, DOUBLE PRECISION : numerical data types with a fixed or variable precision either using *p* bits (for the mantissa) or using *n* as the total number of digits, with *d* digits to the right of the dot
- DATE / TIME : values have the form DATE '<*m*><*m*>/<*d*><*d*>/<*y*><*y*><*y*><*y*>' for specifying a month, day, year, etc. (usually with vendor specific conversion conventions)
- Other variable-length "large" data types: TEXT, BLOB, CLOB, XML, etc.

SQL has type-specific conversions and arithmetic operations:

```
SELECT name
FROM movieStar
WHERE birthdate > DATE '09/28/1980';
```

See, for example, http://www.w3schools.com/sqL/sql_datatypes_general.asp for a detailed description of SQL data types.

Overview of SQL Data Types

Common SQL data types (defined by the ANSI standard):

CHAR(10)	[NOSQLXXXXX]
VARCHAR(10)	[NOSQL'\0']
	[5NOSQL]

- BOOLEAN, BINARY(*n*) : binary types with up to *n* bits
- CHAR(*n*) : fixed-length character string of *n* characters
- VARCHAR(*n*) : variable-length character string of up to *n* characters
- INTEGER / INT, BIGINT, SMALLINT, FLOAT(*p*), NUMERIC(*n*,*d*), DECIMAL(*n*,*d*), REAL, DOUBLE PRECISION : numerical data types with a fixed or variable precision either using *p* bits (for the mantissa) or using *n* as the total number of digits, with *d* digits to the right of the dot
- DATE / TIME : values have the form DATE '<*m*><*m*>/<*d*><*d*>/<*y*><*y*><*y*><*y*>' for specifying a month, day, year, etc. (usually with vendor specific conversion conventions)
- Other variable-length "large" data types: TEXT, BLOB, CLOB, XML, etc.

SQL has type-specific conversions and arithmetic operations:

```
SELECT name
FROM movieStar
WHERE birthdate > DATE '09/28/1980';
```

See, for example, http://www.w3schools.com/sqL/sql_datatypes_general.asp for a detailed description of SQL data types.

Overview of SQL Data Types

Common SQL data types (defined by the ANSI standard):

- BOOLEAN, BINARY(*n*) : binary types with up to *n* bits
- CHAR(*n*) : fixed-length character string of *n* characters
- VARCHAR(*n*) : variable-length character string of up to *n* characters
- INTEGER / INT, BIGINT, SMALLINT, FLOAT(*p*), NUMERIC(*n*,*d*), DECIMAL(*n*,*d*), REAL, DOUBLE PRECISION : numerical data types with a fixed or variable precision either using *p* bits (for the mantissa) or using *n* as the total number of digits, with *d* digits to the right of the dot
- DATE / TIME : values have the form DATE '<*m*><*m*>/<*d*><*d*>/<*y*><*y*><*y*><*y*>' for specifying a month, day, year, etc. (usually with vendor specific conversion conventions)
- Other variable-length "large" data types: TEXT, BLOB, CLOB, XML, etc.

SQL has type-specific conversions and arithmetic operations:

```
SELECT name
FROM movieStar
WHERE birthdate > DATE '09/28/1980';
```

See, for example, http://www.w3schools.com/sqL/sql_datatypes_general.asp for a detailed description of SQL data types.

Primary Keys

The **primary key** of a relation schema *R* is a set of attributes that together uniquely identify every tuple in *R*.

- The projection of the tuples on the primary key may not contain any duplicates.
- Every relation schema may have **at most** one primary key.

SQL provides two syntactic options for defining the primary key of a relation schema:

```
CREATE TABLE MovieStar (
  name CHAR(30) PRIMARY KEY,
  address VARCHAR(255),
  gender CHAR(1),
  birthdate DATE );
```

This is only allowed if the primary key consists of exactly one attribute.

```
CREATE TABLE MovieStar (
  name CHAR(30),
  address VARCHAR(255),
  gender CHAR(1),
  birthdate DATE,
  PRIMARY KEY (name));
```

This allows any combination of attributes to be the primary key.

Defining Multi-Attribute Primary Keys

The following SQL statement defines a multi-attribute primary key that consists of the two attributes title and year.

```
CREATE TABLE Movie (
  title CHAR(30),
  year INTEGER,
  length INTEGER,
  inColor CHAR(1),
  studioName CHAR(50),
  producerCertN INTEGER,
  PRIMARY KEY (title, year));
```

Primary Keys vs. Unique

- If, additionally, we want to express that, also on other projections, the relation instance may not contain duplicates, we may use UNIQUE.
- A UNIQUE constraint may be used similarly to how PRIMARY KEY is used, but there may be more than one UNIQUE constraint per relation schema.

```
CREATE TABLE MovieStar (
  name CHAR(30),
  address VARCHAR(255) UNIQUE,
  gender CHAR(1),
  birthdate DATE,
  PRIMARY KEY (name));
```

vs.

```
CREATE TABLE MovieStar (
  name CHAR(30),
  address VARCHAR(255),
  gender CHAR(1),
  birthdate DATE,
  PRIMARY KEY(name),
  UNIQUE (address));
```

Primary Keys vs. Unique

- If, additionally, we want to express that, also on other projections, the relation instance may not contain duplicates, we may use UNIQUE.
- A UNIQUE constraint may be used similarly to how PRIMARY KEY is used, but there may be more than one UNIQUE constraint per relation schema.

```
CREATE TABLE MovieStar (
  name CHAR(30),
  address VARCHAR(255) UNIQUE,
  gender CHAR(1),
  birthdate DATE,
  PRIMARY KEY (name));
```

vs.

```
CREATE TABLE MovieStar (
  name CHAR(30),
  address VARCHAR(255),
  gender CHAR(1),
  birthdate DATE,
  PRIMARY KEY(name),
  UNIQUE (address));
```

Primary Keys vs. Unique

- If, additionally, we want to express that, also on other projections, the relation instance may not contain duplicates, we may use UNIQUE.
- A UNIQUE constraint may be used similarly to how PRIMARY KEY is used, but there may be more than one UNIQUE constraint per relation schema.

```
CREATE TABLE MovieStar (
  name CHAR(30),
  address VARCHAR(255) UNIQUE,
  gender CHAR(1),
  birthdate DATE,
  PRIMARY KEY (name));
```

vs.

```
CREATE TABLE MovieStar (
  name CHAR(30),
  address VARCHAR(255),
  gender CHAR(1),
  birthdate DATE,
  PRIMARY KEY(name),
  UNIQUE (address));
```

Note:

- NULL values are not allowed under a PRIMARY KEY attribute.
- NULL values are however allowed under a UNIQUE attribute.
- Both conditions also hold for multi-attribute PRIMARY KEY and UNIQUE constraints.

Foreign Keys

A **foreign key** is a **set of attributes** in a relation schema *R* that **together uniquely identify a tuple** in **another relation** *S* that is referenced by *R*.

- The projection of the tuples in *R* onto a foreign key may contain duplicates.
- The referenced attributes must be declared as either UNIQUE or be the PRIMARY KEY of the relation schema *S*.
- Thus, the projection of the tuples in *S* onto the attributes that are referenced by the foreign key may not contain any duplicates.

```
CREATE TABLE MovieExec (
name CHAR(30) UNIQUE,
address VARCHAR(255),
certN INT UNIQUE,
netWorth INT);
```

```
CREATE TABLE Studio (
 name CHAR(30) PRIMARY KEY,
 address VARCHAR(255),
 presCertN INT REFERENCES MovieExec(certN));
```

In general, a relation schema may have multiple foreign keys. Each foreign-key constraint from a relation *R* to a relation *S* defines a N:1 relationship from *R* to *S*.

Special Case:
If the set of attributes $A_1,...,A_n$ in *R* that refer to *S* also is either UNIQUE or the PRIMARY KEY of *R*, then we have a 1:1 relationship.

Foreign Keys

A **foreign key** is a set of attributes in a relation schema *R* that together uniquely identify a tuple in another relation *S* that is referenced by *R*.

- The projection of the tuples in *R* onto a foreign key may contain duplicates.
- The referenced attributes must be declared as either UNIQUE or be the PRIMARY KEY of the relation schema *S*.
- Thus, the projection of the tuples in *S* onto the attributes that are referenced by the foreign key may not contain any duplicates.

```
CREATE TABLE MovieExec (
name CHAR(30) UNIQUE,
address VARCHAR(255),
certN INT UNIQUE,
netWorth INT);
```

```
CREATE TABLE Studio (
 name CHAR(30) PRIMARY KEY,
 address VARCHAR(255),
 presCertN INT REFERENCES MovieExec(certN));
```

Source R Target S

In general, a relation schema may have multiple foreign keys. Each foreign-key constraint from a relation *R* to a relation *S* defines a **N:1 relationship** from *R* to *S*.

Special Case:

If the set of attributes $A_1,...,A_n$ in *R* that refer to *S* also is either UNIQUE or the PRIMARY KEY of *R*, then we have a 1:1 relationship.

Defining Foreign Keys

Once more, SQL provides two syntactic options for defining the foreign key of a relation schema:

```
CREATE TABLE Studio (
  name CHAR(30) PRIMARY KEY,
  address VARCHAR(255),
  presCertN INT REFERENCES MovieExec(certN));
```

This is only allowed if the foreign key consists of exactly one attribute.

```
CREATE TABLE Studio (
  name CHAR(30) PRIMARY KEY,
  address VARCHAR(255),
  presCertN INT,
  FOREIGN KEY (presCertN) REFERENCES MovieExec(certN));
```

This allows any combination of attributes to be a foreign key.

Indexes

An **index** over a set of attributes $A_1,\ldots,A_n$ of a relation schema R is a data structure that allows for efficient lookups of tuples in an instance of R, when given a set of values for $A_1,\ldots,A_n$ as input.

We will refer to a set of values for the index attributes $A_1,\ldots,A_n$ also as a **search key**.

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

If there is an index on the attribute studioName of the relation Movie, then only the tuples that match studioName = 'Disney' will be accessed by the above query, and only those with year = 1990 will also be selected.

We can create an index in SQL as follows:

```
CREATE INDEX StudioIndex ON Movie(studioName);
```

Indexes

An **index** over a set of attributes $A_1,\ldots,A_n$ of a relation schema R is a data structure that allows for efficient lookups of tuples in an instance of R, when given a set of values for $A_1,\ldots,A_n$ as input.

We will refer to a set of values for the index attributes $A_1,\ldots,A_n$ also as a **search key**.

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

If there is an index on the attribute studioName of the relation Movie, then only the tuples that match studioName = 'Disney' will be accessed by the above query, and only those with year = 1990 will also be selected.

We can create an index in SQL as follows:

```
CREATE INDEX StudioIndex ON Movie(studioName);
```

Indexes

An **index** over a set of attributes $A_1,\ldots,A_n$ of a relation schema R is a data structure that allows for efficient lookups of tuples in an instance of R, when given a set of values for $A_1,\ldots,A_n$ as input.

We will refer to a set of values for the index attributes $A_1,\ldots,A_n$ also as a **search key**.

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

If there is an index on the attribute studioName of the relation Movie, then only the tuples that match studioName = 'Disney' will be accessed by the above query, and only those with year = 1990 will also be selected.

We can create an index in SQL as follows:

```
CREATE INDEX StudioIndex ON Movie(studioName);
```

We can drop an index in SQL as follows:

```
DROP INDEX [IF EXISTS] StudioIndex;
```

Indexes

An **index** over a set of attributes $A_1,\ldots,A_n$ of a relation schema R is a data structure that allows for efficient lookups of tuples in an instance of R, when given a set of values for $A_1,\ldots,A_n$ as input.

We will refer to a set of values for the index attributes $A_1,\ldots,A_n$ also as a **search key**.

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

If there is an index on the attribute studioName of the relation Movie, then only the tuples that match studioName = 'Disney' will be accessed by the above query, and only those with year = 1990 will also be selected.

We can create an index in SQL as follows:

```
CREATE INDEX StudioIndex ON Movie(studioName);
```

We can drop an index in SQL as follows:

```
DROP INDEX [IF EXISTS] StudioIndex;
```

Indexes

An **index** over a set of attributes $A_1,\ldots,A_n$ of a relation schema *R* is a data structure that allows for efficient lookups of tuples in an instance of *R*, when given a set of values for $A_1,\ldots,A_n$ as input.

We will refer to a set of values for the index attributes $A_1,\ldots,A_n$ also as a **search key**.

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

Multi-attributed Index

If there is an index on the attribute studioName of the relation Movie, then only the tuples that match studioName = 'Disney' will be accessed by the above query, and only those with year = 1990 will also be selected.

We can create an index in SQL as follows:

```
CREATE INDEX StudioIndex ON Movie(studioName);
```

We can drop an index in SQL as follows:

```
DROP INDEX [IF EXISTS] StudioIndex;
```

Indexes vs. Uniqueness of Values

Multi-attribute indexes can be defined in SQL as follows:

```
CREATE INDEX MovieIndex ON Movie(title, year);
```

```
SELECT *
FROM Movie
WHERE title = 'Terminator' AND year = 1984;
```

Indexes vs. Uniqueness of Values

Multi-attribute indexes can be defined in SQL as follows:

```
CREATE INDEX MovieIndex ON Movie(title, year);
```

Indexes by default do not assume the indexed attribute values (i.e., the search keys) to be unique. If, additionally, we want to express that the projection of a relation onto the indexed attributes may not contain duplicates, we may again use a UNIQUE constraint.

```
CREATE UNIQUE INDEX MovieKeyIndex ON Movie(title, year);
```

Indexes vs. Uniqueness of Values

Multi-attribute indexes can be defined in SQL as follows:

```
CREATE INDEX MovieIndex ON Movie(title, year);
```

Indexes by default do not assume the indexed attribute values (i.e., the search keys) to be unique. If, additionally, we want to express that the projection of a relation onto the indexed attributes may not contain duplicates, we may again use a UNIQUE constraint.

```
CREATE UNIQUE INDEX MovieKeyIndex ON Movie(title, year);
```

Creating an index generally involves a trade-off between speeding up queries; **slowing down insertions, deletions, updates**.

Thus, if large amounts of data are inserted into a relation, it is usually preferable to create all the relation- and database-level constraints **only after the data has been loaded**:

```
SELECT *
FROM Movie
WHERE year = 1984;
```

Indexes vs. Uniqueness of Values

Multi-attribute indexes can be defined in SQL as follows:

```
CREATE INDEX MovieIndex ON Movie(title, year);
```

Indexes by default do not assume the indexed attribute values (i.e., the search keys) to be unique. If, additionally, we want to express that the projection of a relation onto the indexed attributes may not contain duplicates, we may again use a UNIQUE constraint.

```
CREATE UNIQUE INDEX MovieKeyIndex ON Movie(title, year);
```

Creating an index generally involves a trade-off between speeding up queries; slowing down insertions, deletions, updates.

Thus, if large amounts of data are inserted into a relation, it is usually preferable to create all the relation- and database-level constraints only after the data has been loaded:

```
ALTER TABLE Movie ADD CONSTRAINT Movie_PK (title, year) PRIMARY KEY;
ALTER TABLE Studio ADD CONSTRAINT Studio_FK (presCertN) REFERENCES MovieExec(certN);
```

Data Manipulation Language (DML)

Consider once more the same relation schema of Movie:

Movie(title, year, length, inColor, studioName, producerCertN)

Let's insert some example data into our new table:

```
INSERT INTO Movie VALUES ('Skyfall', 2012, 120, 'T', 'Warner', 123);

INSERT INTO Movie VALUES ('QuantumOfSolace', 2008, 134, 'T', 'Warner', 123);

INSERT INTO Movie VALUES ('MightyDucks', 1990, 90, 'T', 'Disney', 456);
```

Data Manipulation Language (DML)

Consider once more the same relation schema of Movie:

Movie(title, year, length, inColor, studioName, producerCertN)

Let's insert some example data into our new table:

```
INSERT INTO Movie VALUES ('Skyfall', 2012, 120, 'T', 'Warner', 123);

INSERT INTO Movie VALUES ('QuantumOfSolace', 2008, 134, 'T', 'Warner', 123);

INSERT INTO Movie VALUES ('MightyDucks', 1990, 90, 'T', 'Disney', 456);
```

Note that all of the following three SQL variations are equivalent:

```
INSERT INTO Movie VALUES ('Skyfall', 2012, 120, 'T', 'Warner', 123);

INSERT INTO Movie (Title, Year, Length, InColor, StudioName, ProducerCertN)
    VALUES ('Skyfall', 2012, 120, 'T', 'Warner', 123);

INSERT INTO Movie (Year, Length, InColor, StudioName, ProducerCertN, Title)
    VALUES (2012, 120, 'T', 'Warner', 123, 'Skyfall');
```

SELECT-FROM-WHERE Statements (DML)

Consider once more the same relation schema of Movie:

Movie(title, year, length, inColor, studioName, producerCertN)

Select all the information of the movies produced by 'Disney' in 1990:

```
SELECT * FROM Movie WHERE studioName = 'Disney' AND year = 1990;
```

➛ The above operation is known as a **selection** in SQL.

Select all titles and lengths of movies produced by 'Disney' in 1990:

```
SELECT title, length FROM Movie WHERE studioName = 'Disney' AND year = 1990;
```

➛ Here, the selection is combined with an additional **projection** in SQL.

Each of the SELECT-FROM-WHERE components is called a **clause** in SQL.

A selection is written as "σ_C" in Relational Algebra, where *C* corresponds to the Boolean condition in the WHERE clause.

A projection is "π_A" in Relational Algebra, where *A* corresponds to the list of attributes in the SELECT clause.

SELECT-FROM-WHERE statements in SQL thus correspond to Select-Project-Join (SPJ) queries in Relational Algebra.

Data Control Language (DCL)

A **database administrator** may grant or revoke access privileges on individual database objects (such as tables, views, etc.) to individual users.

GRANT|REVOKE <p> ON <object> TO <user>;

where <p> is one of the following list of privileges:

CONNECT
SELECT
INSERT
UPDATE
EXECUTE
ALL

Transaction Control Language (TCL)

Multiple subsequent DML commands issued by the same database user form a **transaction**. The database then logs all actions performed within this transaction.

The begin of a transaction can be marked by:

```
BEGIN;
```

Transactions need to be committed into order to persist the data changes to the database:

```
COMMIT;
```

Otherwise the changes are "rolled-back" to the last consistent state of the database:

```
ROLLBACK;
```

Transaction Control Language (TCL)

Multiple subsequent DML commands issued by the same database user form a **transaction**. The database then logs all actions performed within this transaction.

The begin of a transaction can be marked by:

```
BEGIN;
```

Transactions need to be committed into order to persist the data changes to the database:

```
COMMIT;
```

Otherwise the changes are "rolled-back" to the last consistent state of the database:

```
ROLLBACK;
```

Transaction Control Language (TCL) Example

```
CREATE TABLE accounts (
    account_id SERIAL PRIMARY KEY,
    account_name VARCHAR(50),
    balance NUMERIC(10, 2)
);

-- Insert sample data
INSERT INTO accounts (account_name, balance) VALUES
('Alice', 1000.00),('Bob', 500.00);
```

@Bank ATM

Task: Rs.200 from Alice's account to Bob's account

```
BEGIN;

-- Deduct $200 from Alice's account
UPDATE accounts
SET balance = balance - 200
WHERE account_name = 'Alice';

-- Add $200 to Bob's account
UPDATE accounts
SET balance = balance + 200
WHERE account_name = 'Bob';

-- Commit the transaction
COMMIT;
```

Transaction Control Language (TCL) Example

```
CREATE TABLE accounts (
    account_id SERIAL PRIMARY KEY,
    account_name VARCHAR(50),
    balance NUMERIC(10, 2)
);
```

-- Insert sample data

```
INSERT INTO accounts (account_name, balance) VALUES
('Alice', 1000.00),('Bob', 500.00);
```

@Bank ATM

Rollback scenario

```
BEGIN;

-- Deduct $200 from Alice's account
UPDATE accounts
SET balance = balance - 200
WHERE account_name = 'Alice';

-- Simulate an error (e.g., insuff. funds)
DO $$ BEGIN
    IF (SELECT balance FROM accounts WHERE
account_name = 'Alice') < 0 THEN
        RAISE EXCEPTION 'Insufficient funds';
    END IF;
END $$;

-- Add $200 to Bob's account
UPDATE accounts
SET balance = balance + 200
WHERE account_name = 'Bob';

-- Commit the transaction
COMMIT;
```

More DML Examples

Select the title and length of all movies produced by 'Disney' in 1990, and rename the attributes as name and duration:

```
SELECT title AS name, length AS duration
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

Renaming of attributes is written as "ρ_γ" in Relational Algebra, where γ is a bijective (i.e., a one-to-one) mapping from the old to the new attribute names.

Thus, the complete Relational Algebra expression for the above SQL query is:

$\rho_\gamma(\pi_{title,year}(\sigma_{studioName='Disney' \text{ AND } year=1990}(Movie)))$ using

	γ
title	name
length	duration

Select the title and length (in hours) of all movies produced by 'Disney' in 1990, and call the attributes name and lengthInHours (assuming that the length was given in minutes):

```
SELECT title AS name, length/60 AS lengthInHours
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

More SELECT-FROM-WHERE Examples

Select the title and length in hours of all movies produced by 'Disney' in 1990, call the second attribute length, and add an attribute called inHours for which each tuple in the resulting view has the constant value 'hrs.':

```
SELECT title, length/60 AS length, 'hrs.' AS inHours
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

More SELECT-FROM-WHERE Examples

Select the title and length in hours of all movies produced by 'Disney' in 1990, call the second attribute length, and add an attribute called inHours for which each tuple in the resulting view has the constant value 'hrs.':

```
SELECT title, length/60 AS length, 'hrs.' AS inHours
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

Select the title of all movies after '1970' that are filmed in black and white:

```
SELECT title
FROM Movie
WHERE year>1970 AND NOT inColor='T';
```

Ordering Tuples

Queries can be followed by an ORDER BY clause:

ORDER BY <attribute> [ASC | DESC] [, ...]

ORDER BY <sort_expression> [ASC | DESC] [, ...]   ASC is default.

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length/60, title DESC;
```

The above query returns tuples ordered primarily by length and secondarily by title.

Ordering Tuples

Queries can be followed by an ORDER BY clause:

ORDER BY <attribute> [ASC | DESC] [, …]

ORDER BY <sort_expression> [ASC | DESC] [, …]   ASC is default.

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length/60, title DESC;
```

The above query returns tuples ordered primarily by length and secondarily by title.

Length/60	title	. . .
1	b	. . .
1	a	. . .
2	c	. . .
2	c	.
2	b	.
.	.	.

Ordering Tuples

Queries can be followed by an ORDER BY clause:

ORDER BY <attribute> [ASC | DESC] [, ...]

ORDER BY <sort_expression> [ASC | DESC] [, ...] ASC is default.

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length/60, title DESC;
```

The above query returns tuples ordered primarily by length and secondarily by title.

When the order of the attributes is known, we can also use a <list_of_numbers> to denote the column indices (starting with 1).

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990
ORDER BY 3, 1 DESC;
```

Limiting Tuples

We can limit the number of output tuples by using an additional LIMIT clause. This applies to the given order of tuples.

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990
LIMIT 10;
```

Limiting Tuples

We can limit the number of output tuples by using an additional LIMIT clause. This applies to the given order of tuples.

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990
LIMIT 10;
```

Oracle and MySQL additionally provide the built-in attribute ROWNUM which imposes a consecutive counter of tuples to the result of every query.

```
SELECT *
FROM Movie
WHERE year>1970 AND NOT inColor='T' AND ROWNUM <= 10;
```

Comparison & Concatenation

Selection conditions may use >, >=, <, <=, =, <>, +, −, *, / as comparisons and for arithmetic expressions (using common notations for numbers), || (string concatenation), substring(string [from int] [for int]), and Boolean conditions using AND, OR, NOT, (,).

Select the titles of all movies that are made by 'MGM' and were either filmed after 1970 or were less than 90 minutes long:

```
SELECT title
FROM Movie
WHERE (year > 1970 OR length < 90)
    AND studioName = 'MGM';
```

Strings may also be compared by the usual comparison operators <, <=, >, >=, =, <>. Their order is called **lexicographical order**:

A string is a sequence of characters. We assume a total order among all characters in our alphabet (e.g., the ASCII alphabet) to be given.

A string s_1 is "lower than" (<) a string s_2 iff any prefix of length ℓ of s_1 is lower than the corresponding prefix of length ℓ of s_2.

The empty string is "lower than" (<) any other string.

Comparison & Concatenation

Selection conditions may use >, >=, <, <=, =, <>, +, –, *, / as comparisons and for arithmetic expressions (using common notations for numbers), || (string concatenation), substring(string [from int] [for int]), and Boolean conditions using AND, OR, NOT, (,).

Select the titles of all movies that are made by 'MGM' and were either filmed after 1970 or were less than 90 minutes long:

```
SELECT title
FROM Movie
WHERE (year > 1970 OR length < 90)
  AND studioName = 'MGM';
```

'AAB' < 'ABB'

Strings may also be compared by the usual comparison operators <, <=, >, >=, =, <>. Their order is called **lexicographical order**:

A string is a sequence of characters. We assume a total order among all characters in our alphabet (e.g., the ASCII alphabet) to be given.

A string s_1 is "lower than" (<) a string s_2 iff any prefix of length ℓ of s_1 is lower than the corresponding prefix of length ℓ of s_2.

The empty string is "lower than" (<) any other string.

Comparison & Concatenation

Selection conditions may use >, >=, <, <=, =, <>, +, −, *, / as comparisons and for arithmetic expressions (using common notations for numbers), || (string concatenation), substring(string [from int] [for int]), and Boolean conditions using AND, OR, NOT, (,).

Select the titles of all movies that are made by 'MGM' and were either filmed after 1970 or were less than 90 minutes long:

```
SELECT title
FROM Movie
WHERE (year > 1970 OR length < 90)
   AND studioName = 'MGM';
```

'AAB' < 'ABB'

Strings may also be compared by the usual comparison operators <, <=, >, >=, =, <>. Their order is called **lexicographical order**:

A string is a sequence of characters. We assume a total order among all characters in our alphabet (e.g., the ASCII alphabet) to be given.

A string s_1 is "lower than" (<) a string s_2 iff any prefix of length ℓ of s_1 is lower than the corresponding prefix of length ℓ of s_2.

The empty string is "lower than" (<) any other string.

Comparison & Concatenation

Selection conditions may use >, >=, <, <=, =, <>, +, –, *, / as comparisons and for arithmetic expressions (using common notations for numbers), || (string concatenation), substring(string [from int] [for int]), and Boolean conditions using AND, OR, NOT, (,).

Select the titles of all movies that are made by 'MGM' and were either filmed after 1970 or were less than 90 minutes long:

```
SELECT title
FROM Movie
WHERE (year > 1970 OR length < 90)
   AND studioName = 'MGM';
```

'AAB' < 'ABB'

Strings may also be compared by the usual comparison operators <, <=, >, >=, =, <>. Their order is called **lexicographical order**:

A string is a sequence of characters. We assume a total order among all characters in our alphabet (e.g., the ASCII alphabet) to be given.

A string s_1 is "lower than" (<) a string s_2 iff any prefix of length ℓ of s_1 is lower than the corresponding prefix of length ℓ of s_2.

The empty string is "lower than" (<) any other string.

LIKE Operator

Select the title of all movie(s) where the title that starts with 'Star' followed by a single space and then followed by 4 arbitrary characters:

```
SELECT title
FROM Movie
WHERE title LIKE 'Star ____';
```

Allowed wildcard operators are *s* LIKE *p*, *s* NOT LIKE *p*, where *s* is a string and *p* is a pattern, i.e., a string with the optional use of the two special characters;

% matches a sequence of 0 or more characters, and _ matches a single character.

```
SELECT title
FROM Movie
WHERE title LIKE 'Star %';
```

LIKE Operator

Select the title of all movie(s) where the title that starts with 'Star' followed by a single space and then followed by 4 arbitrary characters:

```
SELECT title
FROM Movie
WHERE title LIKE 'Star ____';
```

Allowed wildcard operators are *s* LIKE *p*, *s* NOT LIKE *p*, where *s* is a string and *p* is a pattern, i.e., a string with the optional use of the two special characters;

% matches a sequence of 0 or more characters, and _ matches a single character.

```
SELECT title
FROM Movie
WHERE title LIKE 'Star %';
```

Escape Characters in SQL

A string pattern can be followed by ESCAPE 'x', which indicates that x serves as the escape-character.

'x%' ESCAPE 'x' matches %

'x_' ESCAPE 'x' matches _

'x%%x%' ESCAPE 'x' matches any string that starts and ends with %

Is NULL Equal to NULL?

No. The following query

```
SELECT *
FROM Movie
WHERE NULL = NULL;
```

<u>Note:</u>

Arithmetic operations (+, −, /, *, ||, ...) with NULL always return NULL.

Comparisons (>, <, =, ...) with NULL always return UNKNOWN (which is however shown as NULL, for example, in Postgres).

SQL provides the special built-in predicate IS NULL (and respectively IS NOT NULL) to check for NULL values.

Is NULL Equal to NULL?

No. The following query

```
SELECT *
FROM Movie
WHERE NULL = NULL;
```

returns no tuples.

<u>Note:</u>

Arithmetic operations (+, −, /, *, ||, ...) with NULL always return NULL.

Comparisons (>, <, =, ...) with NULL always return UNKNOWN (which is however shown as NULL, for example, in Postgres).

SQL provides the special built-in predicate IS NULL (and respectively IS NOT NULL) to check for NULL values.

Joins

Joins in SQL involve more than one relation in the FROM clause.

Consider the following two relation schemas.

Movie(title, year, length, inColor, studioName, producerCertN)

MovieExec(name, address, certN, netWorth)

Select the name of the producers of 'Star Wars':

```
SELECT name FROM Movie, MovieExec
WHERE title = 'Star Wars' AND producerCertN = certN;
```

➛ This results in all combinations of tuples that fulfill the WHERE condition.

Joins

Joins in SQL involve more than one relation in the FROM clause.

Consider the following two relation schemas.

Movie(title, year, length, inColor, studioName, producerCertN)

MovieExec(name, address, certN, netWorth)

Select the name of the producers of 'Star Wars':

```
SELECT name FROM Movie, MovieExec
WHERE title = 'Star Wars' AND producerCertN = certN;
```

➛ This results in all combinations of tuples that fulfill the WHERE condition.

This so-called **theta-join** is written as "$\bowtie_C$" in Relational Algebra, where *C* corresponds to the join condition (here: producerCertN = certN) in the WHERE clause.

But what about the ambiguity of attribute names?

Select all pairs of stars and executives that have the same address:

```
SELECT MovieStar.name, MovieExec.name FROM MovieStar, MovieExec
WHERE MovieStar.address = MovieExec.address;
```

The notation <relation>.<attribute> is permissible, even when there is no ambiguity.

Self-Joins

What to write if we need a same relation several times in the FROM clause?

We use **tuple variables** (aka. "table aliases").

Select all pairs of stars that have the same address:

```
SELECT Star1.name, Star2.name
FROM MovieStar AS Star1, MovieStar AS Star2
WHERE Star1.address = Star2.address;
```

or:

```
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address;
```

Self-Joins

What to write if we need a same relation several times in the FROM clause?

We use **tuple variables** (aka. "table aliases").

Select all pairs of stars that have the same address:

```
SELECT Star1.name, Star2.name
FROM MovieStar AS Star1, MovieStar AS Star2
WHERE Star1.address = Star2.address;
```

or:

```
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address;
```

Here, Star1 and Star2 are so-called tuple variables. However, if we get one such pair of stars, say (*a*,*b*), then we also get all the *reflexive* and *symmetric* cases (*b*,*a*), (*a*,*a*) and (*b*,*b*) as results. We can remove these redundancies by using the following trick:

```
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address AND Star1.name < Star2.name;
```

Nested Queries in the FROM Clause

SQL is an expression language. Subqueries may be nested (almost) anywhere inside an embracing SQL expression.

Select the name of all producers of 'Star Wars':

```
SELECT name
FROM MovieExec, (SELECT producerCertN  prod
                 FROM  Movie WHERE title = 'Star Wars' ) Temp
WHERE Temp.prod = certN;
```

Nested Queries in the FROM Clause

SQL is an expression language. Subqueries may be nested (almost) anywhere inside an embracing SQL expression.

Select the name of all producers of 'Star Wars':

```
SELECT name
FROM MovieExec, (SELECT producerCertN  prod
                 FROM  Movie WHERE title = 'Star Wars' ) Temp
WHERE Temp.prod = certN;
```

Notice that

```
SELECT A, E
FROM (SELECT A, C, B, D, E FROM R
      WHERE B = 'x')
WHERE A > 'c';
```

is equivalent to:

```
SELECT A, E
FROM R
WHERE B = 'x' AND A > 'c';
```

UNION; INTERSECT; EXCEPT (I)

Consider the following two relation schemas.

MovieExec(name, address, certN, netWorth)

MovieStar(name, address, gender, birthdate)

Select the names and addresses of all female movie stars who are also movie executives with the same address and with a net worth over '$1M':

```
SELECT name, address
FROM MovieStar
WHERE gender = 'F'
   INTERSECT
SELECT name, address
FROM MovieExec
WHERE netWorth > 1000000;
```

UNION; INTERSECT; EXCEPT (I)

Consider the following two relation schemas.

MovieExec(name, address, certN, netWorth)

MovieStar(name, address, gender, birthdate)

Select the names and addresses of all female movie stars who are also movie executives with the same address and with a net worth over '$1M':

```
SELECT name, address
FROM MovieStar
WHERE gender = 'F'
  INTERSECT
SELECT name, address
FROM MovieExec
WHERE netWorth > 1000000;
```

Select the names and addresses of all movie stars who are **not** also movie executives with the same address:

```
SELECT name, address FROM MovieStar
  EXCEPT SELECT name, address FROM MovieExec;
```

(in Oracle: use MINUS instead of EXCEPT)

UNION; INTERSECT; EXCEPT (II)

Consider the following two relation schemas.

MovieExec(name, address, certN, netWorth)

MovieStar(name, address, gender, birthdate)

Select the titles and years of all movies that appeared in either the Movie or in the StarsIn relation:

```
SELECT title, year FROM Movie
  UNION
SELECT movieTitle AS title, movieYear AS year FROM StarsIn;
```

<u>Notice:</u> data types of the attributes

According to the SQL standard, the projected by the SELECT clauses of both subqueries have to be equal.

SQL also requires them occur be in the same order in both of the SELECT clauses.

The attribute names however **do not** need to match; if they are different in each of the subqueries, the attribute names of the first subquery are applied to the result.

UNION ALL; INTERSECT ALL; EXCEPT ALL

Each of UNION, INTERSECT and EXCEPT automatically eliminate duplicates to enforce a set semantics. If we do not want duplicates to be eliminated, we may use

UNION ALL, INTERSECT ALL, EXCEPT ALL

instead.

R **UNION ALL** *S*: Tuple *t* appears the **sum of times** it appears in both *R* and *S*.

R **INTERSECT ALL** *S*: Tuple *t* appears the **minimum amount of times** it appears in *R* and *S*.

R **EXCEPT ALL** *S* : Tuple *t* appears the **number of times** it appears in *R* **minus** the **number of times** it appears in *S*, and **at least 0** times.

UNION ALL; INTERSECT ALL; EXCEPT ALL

Each of UNION, INTERSECT and EXCEPT automatically eliminate duplicates to enforce a set semantics. If we do not want duplicates to be eliminated, we may use

UNION ALL, INTERSECT ALL, EXCEPT ALL

instead.

R **UNION ALL** *S*: Tuple *t* appears the **sum of times** it appears in both *R* and *S*.

R **INTERSECT ALL** *S*: Tuple *t* appears the **minimum amount of times** it appears in *R* and *S*.

R **EXCEPT ALL** *S* : Tuple *t* appears the **number of times** it appears in *R* **minus** the **number of times** it appears in *S*, and **at least 0** times.

The following three queries yield the same result, each by eliminating duplicates in a different way.

- (SELECT title FROM Movie) **INTERSECT** (SELECT title FROM Movie);
- (SELECT title FROM Movie) **UNION** (SELECT title FROM Movie);
- SELECT **DISTINCT** title FROM Movie;

<u>Notice:</u> INTERSECT ALL and EXCEPT ALL are not supported in Oracle but in PostgreSQL.

Explicit Duplicate Elimination

In order to eliminate duplicates in SQL (thus enforcing a set semantics), we may use the modifier DISTINCT.

Even if there are no duplicates in the input relations, duplicates may arise due to a projection (here in combination with a join):

```
SELECT DISTINCT name
 FROM MovieExec, Movie, StarsIn
 WHERE      certN = ProducerCertN AND title = movieTitle AND
    year = movieYear AND starName = 'Harrison Ford';
```

In the next query, we do not need to use DISTINCT, since there are no duplicates that could arise due to a join with a subsequent projection:

```
SELECT name
 FROM MovieExec
 WHERE certN IN (SELECT producerCertN
                 FROM Movie
                 WHERE title IN (SELECT movieTitle
                                 FROM StarsIn
                                 WHERE starName = 'Harrison Ford' AND year = movieYear));
```

Basically, IN would see if {1} IN {1,3,4}

More Ways to Connect Subqueries in the WHERE Clause

Assume that *s* is a scalar value or a single attribute; *R* is a unary relation whose tuples are of the same type.

SQL provides the following ways to connect subqueries in the WHERE clause:

- EXISTS (*R*) iff relation *R* is not empty
- *s* IN (*R*) iff *s* is in relation *R*, detects the membership in a bag
- *s* $>$ ALL (*R*) iff *s* is greater than *all* the values in the relation *R* ($<$, $<=$, $>=$, $=$, $<>$)
- *s* $>$ ANY (*R*) iff *s* is greater than *at least one* value in the relation *R* ($<$, $<=$, $>=$, $=$, $<>$)
- Also: NOT EXISTS (*R*), *s* NOT IN (*R*), NOT *s* $>$ ALL (*R*) , NOT *s* $>$ ANY (*R*)

More Ways to Connect Subqueries in the WHERE Clause

Assume that *s* is a scalar value or a single attribute; *R* is a unary relation whose tuples are of the same type.

SQL provides the following ways to connect subqueries in the WHERE clause:

- EXISTS (*R*) iff relation *R* is not empty
- *s* IN (*R*) iff *s* is in relation *R*, detects the membership in a bag
- *s* $>$ ALL (*R*) iff *s* is greater than *all* the values in the relation *R* ($<$, $<=$, $>=$, $=$, $<>$)
- *s* $>$ ANY (*R*) iff *s* is greater than *at least one* value in the relation *R* ($<$, $<=$, $>=$, $=$, $<>$)
- Also: NOT EXISTS (*R*), *s* NOT IN (*R*), NOT *s* $>$ ALL (*R*) , NOT *s* $>$ ANY (*R*)

Note:

- Comparisons of ∅ with ANY return FALSE; comparisons of ∅ with ALL return TRUE.
- 'NOT *s* $=$ ANY' is equivalent to '*s* $<>$ ALL' (incl. the case when the subquery returns ∅).

More Ways to Connect Subqueries in the WHERE Clause

Assume that *s* is a scalar value or a single attribute; *R* is a unary relation whose tuples are of the same type.

SQL provides the following ways to connect subqueries in the WHERE clause:

- EXISTS (*R*) iff relation *R* is not empty
- *s* IN (*R*) iff *s* is in relation *R*, detects the membership in a bag
- *s* $>$ ALL (*R*) iff *s* is greater than *all* the values in the relation *R* ($<$, $<=$, $>=$, $=$, $<>$)
- *s* $>$ ANY (*R*) iff *s* is greater than *at least one* value in the relation *R* ($<$, $<=$, $>=$, $=$, $<>$)
- Also: NOT EXISTS (*R*), *s* NOT IN (*R*), NOT *s* $>$ ALL (*R*) , NOT *s* $>$ ANY (*R*)

Note:

- Comparisons of Ø with ANY return FALSE; comparisons of Ø with ALL return TRUE.
- 'NOT *s* $=$ ANY' is equivalent to '*s* $<>$ ALL' (incl. the case when the subquery returns Ø).
- Non-scalar values are not allowed in ANSI SQL, but are allowed in Oracle and Postgres:

```
SELECT *
FROM R
WHERE (R.A, R.B) <> ALL
   (SELECT S.A, S.B FROM S);
```

is equivalent to

```
SELECT *
FROM R
WHERE R.A <> ALL
   (SELECT S.A FROM S WHERE R.B = S.B);
```

Aggregations

An **aggregation** is a database operation that forms a single value from a bag of values.

In SQL, the standard aggregations include:

- MIN, MAX, COUNT, SUM, AVG

Duplicates need to explicitly be eliminated by using DISTINCT.

NULL values by default are not considered for the numerical aggregation functions (except for COUNT(*), which counts also the number of tuples with NULL values in its input).

Aggregations

An **aggregation** is a database operation that forms a single value from a bag of values.

In SQL, the standard aggregations include:

- MIN, MAX, COUNT, SUM, AVG

Duplicates need to explicitly be eliminated by using DISTINCT.

NULL values by default are not considered for the numerical aggregation functions (except for COUNT(*), which counts also the number of tuples with NULL values in its input).

Consider the following relation schema.

MovieExec(name, address, certN, netWorth)

Select the average net worth of all movie executives:

```
SELECT AVG(netWorth) FROM MovieExec;
```

Select the number of tuples in the relation MovieExec:

```
SELECT COUNT(*) FROM MovieExec;
```

Select the number of different names of executives in MovieExec:

```
SELECT COUNT(DISTINCT name) FROM MovieExec;
```

Aggregations & Grouping

Consider the following relation schema.

Movie(title, year, length, inColor, studioName, producerCertN)

Select the sum of the lengths of all movies that were produced by each distinct studio:

```
SELECT studioName, SUM(length) FROM Movie GROUP BY studioName;
```

Aggregations & Grouping

Consider the following relation schema.

Movie(title, year, length, inColor, studioName, producerCertN)

Select the sum of the lengths of all movies that were produced by each distinct studio:

```
SELECT studioName, SUM(length) FROM Movie GROUP BY studioName;
```

First group all the tuples per studio name, then calculate the sum for each such group and create one result tuple for each distinct studio.

```
SELECT SUM(length) FROM Movie GROUP BY studioName;
```

➛ This query however only yields the aggregated movie lengths, but loses the studio names.

Aggregations & Grouping

Consider the following relation schema.

Movie(title, year, length, inColor, studioName, producerCertN)

Select the sum of the lengths of all movies that were produced by each distinct studio:

```
SELECT studioName, SUM(length) FROM Movie GROUP BY studioName;
```

First group all the tuples per studio name, then calculate the sum for each such group and create one result tuple for each distinct studio.

```
SELECT SUM(length) FROM Movie GROUP BY studioName;
```

➢ This query however only yields the aggregated movie lengths, but loses the studio names.

Generally, if there is an aggregation in the SELECT clause, then the non-aggregated attributes in the SELECT clause must appear also in the GROUP BY clause.

```
SELECT inColor, SUM(length)
FROM Movie
GROUP BY studioName; (* NOT VALID SQL *)
```

```
SELECT SUM(length)
FROM Movie; (* VALID SQL *)
```

More about Aggregations & Grouping

The following query

```
SELECT title
FROM Movie GROUP BY title;
```

yields the same result as:

```
SELECT DISTINCT title
FROM Movie;
```

Consider the following two relation schemas.

Movie(title, year, length, inColor, studioName, producerCertN)

MovieExec(name, address, certN, netWorth)

Select, for each producer, the total length of films produced by that producer:

```
SELECT name, SUM(length)
FROM MovieExec, Movie WHERE producerCertN = certN
GROUP BY name;
```

Select, for each producer with a net worth of more than $1M, the total length of films produced by that producer:

```
SELECT name, SUM(length)
FROM MovieExec, Movie WHERE producerCertN = certN AND netWorth >= 1000000
GROUP BY name;
```

Filtering Conditions for Groups

The HAVING clause allows us to post-filter groups by a Boolean condition.

Select, for each producer with a net worth of more than $1M and who made at least one film prior to '1930', the total length of films produced:

```
SELECT name, SUM(length)
FROM MovieExec, Movie
WHERE producerCertN = certN AND netWorth >= 1000000
GROUP BY name
HAVING MIN(year) < 1930;
```

Each unaggregated attribute in the HAVING clause must appear in the GROUP BY clause.

Filtering Conditions for Groups

The HAVING clause allows us to post-filter groups by a Boolean condition.

Select, for each producer with a net worth of more than $1M and who made at least one film prior to '1930', the total length of films produced:

```
SELECT name, SUM(length)
FROM MovieExec, Movie
WHERE producerCertN = certN AND netWorth >= 1000000
GROUP BY name
HAVING MIN(year) < 1930;
```

Each unaggregated attribute in the HAVING clause must appear in the GROUP BY clause.

Evaluation order of SQL clauses: SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY

1. Evaluate the FROM and the WHERE clause;
2. group the tuples according to the GROUP BY clause;
3. select the groups according to the HAVING clause;
4. produce the result according to the SELECT and the ORDER BY clause.

More reading..

More About Joins in SQL

The JOIN ON operator provides the full syntax for a binary join in SQL. It is equivalent to writing two relations in the FROM clause of the query with a corresponding join condition in the WHERE clause. JOIN ON corresponds to the theta-join ("θ_C") in Relational Algebra.

The result of the following query

```
SELECT * FROM Movie JOIN StarsIn ON title = movieTitle AND year = movieYear;
```

has 9 attributes called:

title, year, length, inColor, studioName, producerCertN, movieTitle, movieYear, starName

An attribute may be preceded (and must be in the case of ambiguity) by the name of its relation, followed by a dot.

Consider the following two relation schemas.

MovieStar(name, address, gender, birthdate)

MovieExec(name, address, certN, netWorth)

The following query returns two name and address attributes for each result tuple:

```
SELECT MovieStar.name, MovieExec.name, MovieStar.address, MovieExec.address
FROM MovieStar JOIN MovieExec USING (name, address)
WHERE MovieStar.name LIKE 'J%';
```

Natural, Cross & Outer Joins

The NATURAL JOIN operator corresponds to the natural join ("$\bowtie$") in Relational Algebra.

Consider the following two relation schemas.

MovieStar(name, address, gender, birthdate)

MovieExec(name, address, certN, netWorth)

Then, the following two queries are equivalent:

```
SELECT * FROM MovieStar NATURAL JOIN MovieExec;
SELECT MovieStar.name, MovieStar.address, gender, birthdate, certN, netWorth
  FROM MovieStar, MovieExec WHERE MovieStar.name = MovieExec.name
  AND MovieStar.address = MovieExec.address;
```

The result of both queries has 6 attributes, namely:

name, address, gender, birthdate, certN, netWorth

A CROSS JOIN operator corresponds to the Cartesian product ("$\times$") in Relational Algebra.

An OUTER JOIN operator introduces NULL values to augment tuples that have no matching join partner in the other relation:

[NATURAL] FULL|LEFT|RIGHT OUTER JOIN

Natural & Outer Join Examples (I)

Consider the following two relation instances as input:

R (A B)

A	B
a	b
d	e

S (B C)

B	C
b	c
f	g

Then, we have:

(R NATURAL FULL OUTER JOIN S)

A	B	C
a	b	c
d	e	NULL
NULL	f	g

(R NATURAL LEFT OUTER JOIN S)

A	B	C
a	b	c
d	e	NULL

(R NATURAL RIGHT OUTER JOIN S)

A	B	C
a	b	c
NULL	f	g

Natural & Outer Join Examples (II)

Consider the following two relation instances as input:

R (A B)

A	B
a	b
d	e

S (B C)

B	C
b	c
f	g

Then, we have:

(R FULL OUTER JOIN S ON R.A > S.C)

A	B_1	B_2	C
d	e	b	c
a	b	NULL	NULL
NULL	NULL	f	g

(R LEFT OUTER JOIN S ON R.A > S.C)

A	B_1	B_2	C
d	e	b	c
a	b	NULL	NULL

(R RIGHT OUTER JOIN S ON R.A > S.C)

A	B_1	B_2	C
d	e	b	c
NULL	NULL	f	g

Sliding-Window Operator (introduced ANSI/ISO SQL:2003, revised in SQL:2011)

Consider the following relation schema.

StarsIn(movieTitle, movieYear, starName)

General syntax of the OVER operator (embedded into SELECT clause):

<aggregation function> OVER
([PARTITION BY <attribute_list>]
[ORDER BY <attribute [ASC|DESC] list>])

Special Case 1: no partitioning, just create one fixed window for all tuples in StarsIn:

```
SELECT movieTitle, movieYear, starName, count(*) OVER () FROM StarsIn;
```

Special Case 2: still no partitioning, but create a sliding window of all tuples in StarsIn, ordered up to the current tuple:

```
SELECT movieTitle, movieYear, starName, count(*) OVER (ORDER BY
        movieTitle, movieYear, starName) FROM StarsIn;
```

General Case: use both partitioning and create a sliding window of all tuples in StarsIn, ordered up to the current tuple:

```
SELECT movieTitle, movieYear, starName, count(*) OVER (PARTITION BY starName ORDER BY
        movieTitle, movieYear, starName) FROM StarsIn;
```

Sliding-Window Operator with Additional Ranking

The rank() operator returns the position (i.e., "rank") of the current tuple within the given sliding window:

Here both count(*) and rank() give the same results:

```
SELECT movieTitle, movieYear, starName,
  count(*) OVER (PARTITION BY starName ORDER BY movieTitle, movieYear, starName),
  rank() OVER (PARTITION BY starName ORDER BY movieTitle, movieYear, starName)
FROM StarsIn;
```

Here, however, we can see that count(*) is a local counter that is restarted at each partition, while rank() remains a global counter:

```
SELECT movieTitle, movieYear, starName,
  count(*) OVER (PARTITION BY starName ORDER BY movieTitle, movieYear, starName),
  rank() OVER (ORDER BY movieTitle, movieYear, starName)
FROM StarsIn;
```

Some More Advanced SQL Examples (I)

Consider once more the following three relation schemas (cf. Chapter II, Slide 19).

Likes(drinker, beer); Serves(bar, beer); Visits(drinker, bar)

Select the distinct drinkers that like *any* beer that is served in *any* bar that the visit.

```
SELECT DISTINCT Likes.drinker
FROM Likes, Serves, Visits
WHERE Likes.Beer = Serves.Beer AND Serves.Bar = Visits.Bar;
```

Select the distinct drinkers that like *any* beer that is served in the 'Urban' bar.

```
SELECT DISTINCT Likes.drinker
FROM Likes, Serves
WHERE Likes.Beer = Serves.Beer AND Serves.Bar = 'Urban';
```

Select the distinct drinkers that like *all* the beers that are served in the 'Urban' bar.

```
SELECT Drinker FROM Likes EXCEPT
  SELECT Temp.Drinker FROM (
    SELECT L.Drinker, S.Beer FROM Likes L, Serves S WHERE S.Bar = 'Urban' EXCEPT
      SELECT Drinker, Beer FROM Likes) Temp;
```

Some More Advanced SQL Examples (II)

Consider once more the following three relation schemas (cf. Chapter II, Slide 19).

Likes(<u>drinker</u>, <u>beer</u>); Serves(<u>bar</u>, <u>beer</u>); Visits(<u>drinker</u>, <u>bar</u>)

Select the distinct drinkers that like *exactly* the beers that are served in the 'Urban' bar.

```
SELECT Drinker FROM Likes
  EXCEPT
SELECT Temp1.Drinker FROM (SELECT L.Drinker, S.Beer FROM Likes L, Serves S WHERE    S.Bar = 'Urban' EXCEPT SELECT Drinker, Beer FROM Likes) Temp1
  EXCEPT
SELECT Temp2.Drinker FROM (SELECT Drinker, Beer FROM Likes EXCEPT SELECT Likes.Drinker,      Serves.Beer FROM Likes, Serves WHERE Serves.Bar='Urban') Temp2;
```

Select the distinct pairs of beers that are served in *two different* bars.

```
SELECT DISTINCT S1.Beer, S2.Beer FROM Serves S1, Serves S2 WHERE S1.Bar != S2.Bar;
```

Select the distinct pairs of beers that are *not* served in a *common* bar.

```
SELECT S1.Beer, S2.Beer FROM Serves S1, Serves S2
  EXCEPT
SELECT S1.Beer, S2.Beer FROM Serves S1, Serves S2 WHERE S1.Bar = S2.Bar;
```

Division in SQL

The **division** operator (":") is the inverse of the Cartesian product ("×") in Relational Algebra.

Select the names of stars that act in *all* the movies with the title 'Terminator':

```
SELECT name FROM StarsIn
  EXCEPT
SELECT C.starName FROM (
   SELECT A.starName, B.title, B.year FROM StarsIn A, Movie B
   WHERE B.title = 'Terminator'
     EXCEPT
   SELECT starName, movieTitle, movieYear FROM StarsIn) C;
```

Consider two relations schemas R(A,B) and S(B).

In general, the division R(A,B) : S(B) is given by:

```
SELECT A FROM R
  EXCEPT
SELECT A FROM (
   SELECT R.A, S.B FROM R, S
     EXCEPT
   SELECT A, B FROM R);
```

In Relational Algebra:

$$R : S \equiv \pi_{\Omega R-\Omega S}(R) - \pi_{\Omega R-\Omega S}((\pi_{\Omega R-\Omega S}(R) \times S) - R)$$

Notice that the set-difference EXCEPT removes duplicates from *R* both in SQL and in Relational Algebra.

Set Equivalence in SQL

An actual operator for **set equivalence** is not built into SQL. We need to first check whether the first set is not a subset of the latter, and then whether the latter set is not a subset of the former. If their union is empty, both sets must be the same (and/or also be empty).

The modifier ALL here ensures that the query also works in the presence of duplicates (i.e., for bags of tuples rather than sets).

Select all distinct pairs of actors that act in exactly the same sets of movies:

```
SELECT DISTINCT S1.starName, S2.starName
FROM StarsIn S1, StarsIn S2
WHERE S1.starName < S2. starName AND NOT exists (
    (SELECT movieTitle, movieYear FROM StarsIn WHERE starName=S1.starName
        EXCEPT ALL
     SELECT movieTitle, movieYear FROM StarsIn WHERE starName=S2.starName)
  UNION ALL
    (SELECT movieTitle, movieYear FROM StarsIn WHERE starName=S2.starName
        EXCEPT ALL
     SELECT movieTitle, movieYear FROM StarsIn WHERE starName=S1.starName));
```

Median in SQL

The **median** of a sorted list of length ℓ is its middle element (i.e., the element that splits the upper and lower parts of the list into two equal sizes).

If there is *an uneven number of elements*, then choose the element at position $(\ell+1)/2$.

If there is an even number of elements, then choose the element at position $\ell/2 + 1$.

➛ Use position ceiling$((\ell + 0.1)/2)$ in both cases.

```
SELECT M.title, M.year, M.N from (
   SELECT title, year,
       count(*) OVER() AS L,
       count(*) OVER(ORDER BY title, year) AS N
   FROM Movie) AS M
WHERE N = ceiling((L+0.1)/2);
```

Do You Know SQL?

What is the difference between

```
SELECT B
FROM R
WHERE A<10 OR A>=10;
```

and simply:

```
SELECT B
FROM R;
```

R

A	B
5	20
10	30
20	40
…	…

Do You Know SQL?

What about these?

```
SELECT A
FROM R, S
WHERE R.B = S.B;

SELECT A
FROM R
WHERE B IN (SELECT B FROM S);
```

R

A	B
5	20
10	30
20	40
…	…

2. Database Modifications: Insertions (part of DML)

Database insertions can either be done manually by inserting one tuple at a time (by providing a set of values for a relation's attributes), or one may insert many tuples directly from a subquery.

INSERT INTO <relation> [($A_1,\ldots,A_n$)] VALUES ($v_1,\ldots,v_n$);

INSERT INTO <relation> [($A_1,\ldots,A_n$)] <subquery>;

In both cases, the data types of the relation's attributes and the new values must coincide.

Consider the following two relation schemas.

Studio(name, address, presCertN)

StarsIn(movieTitle, movieYear, starName)

We may manually insert new tuples into a table as follows:

```
INSERT INTO StarsIn(movieTitle, movieYear, starName)
    VALUES ('Skyfall', 2012, 'Craig');
INSERT INTO StarsIn VALUES ('Skyfall', 2012, 'Craig');
```

The default order of attributes is the one defined by the CREATE TABLE statement.

```
INSERT INTO Studio(name, presCertN) VALUES ('Broccoli',  235);
```

Here, the tuple ('Broccoli', NULL, 235) is inserted.

Insertions from a Subquery

Consider the following two relation schemas.

Movie(title, year, length, inColor, studioName, producerCertN)

Studio(name, address, presCertN)

Insert into the relation instance of Studio all the movie studio names that are mentioned in the relation Movie but do not yet appear in Studio:

```
INSERT INTO Studio(name)
   SELECT DISTINCT studioName
   FROM Movie
   WHERE studioName NOT IN (
       SELECT name
       FROM Studio);
```

This results in NULL values under the attributes address, presCertN.

Notice, again, that the order of attributes in the SELECT clause is important and that the data types of the attributes must match.

Bulkloading Data from Files (Postgres-specific COPY command)

Different database vendors offer different tools to **bulkload** large amounts of data from files into a previously defined relation schema.

Postgres provides the following COPY command

```
COPY <relation> [ ( <column> [, ...] ) ] FROM { '<file>' | STDIN }
  [ [ WITH ] ( <option> [, ...] ) ]
```

where <option> may be one of:

```
DELIMITER '<delimiter_character>'
NULL '<null_string>'
HEADER [ TRUE | FALSE ]
QUOTE '<quote_character>'
ESCAPE '<escape_character>'
ENCODING '<encoding_name>'
```

For example, to load a TSV file /tmp/lineitem.tsv into the relation LINEITEM, we may issue the following command in the Postgres client shell:

```
COPY lineitem FROM '/tmp/lineitem.tsv' WITH DELIMITER '\t';
```

Deletions (part of DML)

The WHERE clause of a DELETE statement has the same functionality as in a SELECT statement: all tuples for which the Boolean condition specified in the WHERE clause evaluates to TRUE are deleted.

DELETE FROM <relation> WHERE <condition>;

Consider the following three relation schemas.

Movie(title, year, length, inColor, studioName, producerCertN)

StarsIn(movieTitle, movieYear, starName)

MovieExec(name, address, certN, netWorth)

Delete from the relation StarsIn all tuples where 'Craig' was a star in 'Skyfall':

```
DELETE FROM StarsIn
WHERE movieTitle = 'Skyfall' AND starName = 'Craig';
```

Delete from MovieExec all those movie executives whose net worth is less than $1M and who are not a president of a movie:

```
DELETE FROM MovieExec
WHERE netWorth < 1000000 AND certN NOT IN (
    SELECT producerCertN FROM Movie);
```

Updates (part of DML)

Updates allow us to change individual values of a tuple. Semantics of WHERE is as before.

UPDATE <relation> SET <new-value assignments> WHERE <condition>;

Consider the following two relation schemas.

MovieExec(name, address, certN, netWorth)

Studio(name, address, presCertN)

Update the relation MovieExec by prepending the title 'Pres.' in front of every movie executive who is president of a studio:

```
UPDATE MovieExec SET name = 'Pres. ' || name
WHERE certN IN (SELECT presCertN FROM Studio);
```

Add $5M to the net worth of every movie executive that lives in 'Hollywood' and change the address to 'Luxembourg':

```
UPDATE MovieExec SET netWorth = netWorth + 5000000, address = 'Luxembourg'
WHERE address = 'Hollywood';
```

Important Note: For both DELETE and UPDATE, the relation that is being modified must not occur within a subquery of the WHERE clause!

Note on Insertions, Deletions & Updates

INSERT inserts one tuple at a time, even when creating duplicates (and no PRIMARY KEY or UNIQUE constraints are violated).

DELETE deletes all the tuples that satisfy the condition in the WHERE clause.

Thus, there is no (easy) way to delete a single tuple of a relation that contains duplicates.

That is,

```
INSERT INTO StarsIn
   VALUES ('Skyfall', 2012, 'Craig');

DELETE FROM StarsIn
    WHERE movieTitle = 'Skyfall' AND movieYear = 2012
   AND starName = 'Craig';
```

may result in a relation that is different from the input relation StarsIn before the insertion was issued!

Schema Modifications (part of DDL)

Recall the declaration of a relation schema:

```
CREATE TABLE MovieStar (
  name CHAR(30),
  address VARCHAR(255),
  gender CHAR(1),
  birthdate DATE );
```

Relation schema modification:

```
ALTER TABLE MovieStar ADD phone CHAR(16);
ALTER TABLE MovieStar DROP birthdate;
```

(not supported by Oracle but by Postgres)

Setting default values:

```
CREATE TABLE MovieStar (
  name CHAR(30),
  address VARCHAR(255),
  gender CHAR(1) DEFAULT '?',
  birthdate DATE DEFAULT '01 jan 1000');

ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT 'unlisted';
```

Attribute Domains

A DOMAIN and DEFAULT in SQL restrict the values that we may assign to an attribute; these usually specify a subset of values from a regular SQL data type.

```
CREATE DOMAIN AddressDomain AS VARCHAR(255);
CREATE DOMAIN GenderDomain AS CHAR(1) DEFAULT '?';
CREATE TABLE MovieStar (
    name CHAR(30),
    address AddressDomain,
    gender GenderDomain,
    birthdate DATE DEFAULT '00/00/0000');
```

(not supported by Oracle, but by Postgres!)

SQL however does not define what happens with conflicting defaults.

```
ALTER DOMAIN GenderDomain AS CHAR(1) DEFAULT '*';
```

Only the default value can be altered.

```
DROP DOMAIN [IF EXISTS] GenderDomain;
```

(Altering types/default values of domains is not supported by Postgres.)

The attributes already defined using this domain will continue to have the same type and default as they had before dropping the domain.

Views

Views are relations that are not physically stored by the DBMS. They are defined by another query. Views themselves can be queried and (in some cases) also be modified.

```
CREATE VIEW <view_name> AS <view_definition>;
DROP VIEW [IF EXISTS] <view_name>;
```

Consider the following relation schema.

Movie(title, year, length, inColor, studioName, producerCertN)

Create a new view called ParamountMovie by using Movie as a base relation:

```
CREATE VIEW ParamountMovie AS
  SELECT title, year, studioName
  FROM Movie
  WHERE studioName = 'Paramount';
```

The view ParamountMovie is not actually physically stored by the DBMS, but queried over its base relation(s):

```
SELECT title, year
FROM ParamountMovie
WHERE NOT studioName = 'Paramount';
```

Querying Views (I)

Consider the following view definition.

```
CREATE VIEW ParamountMovie AS
  SELECT title, year
  FROM Movie
  WHERE studioName = 'Paramount';
```

This query over the view

```
SELECT title
FROM ParamountMovie
WHERE year = 1979;
```

is internally transformed by the DBMS into the following query over the base relation:

```
SELECT title
FROM Movie
WHERE year = 1979 AND studioName = 'Paramount';
```

Querying Views (II)

Consider the following relation schema.

StarsIn(movieTitle, movieYear, starName)

This join query over the view and the above relation

```
SELECT DISTINCT starName
FROM ParamountMovie, StarsIn
WHERE title = movieTitle AND year = movieYear;
```

is transformed by the DBMS into:

```
SELECT DISTINCT starName
FROM Movie, StarsIn
WHERE studioName = 'Paramount' AND title = movieTitle AND year = movieYear;
```

Querying Views (III)

Consider the following two relation schemas.

Movie(title, year, length, inColor, studioName, producerCertN)

MovieExec(name, address, certN, netWorth)

Consider the following view definition.

```
CREATE VIEW MovieProd AS
   SELECT title, name
   FROM Movie, MovieExec
  WHERE producerCertN = certN;
```

This query

```
SELECT name, COUNT(*) FROM MovieProd GROUP BY name;
```

is transformed by the DBMS into:

```
SELECT name, COUNT(*)
FROM Movie, MovieExec
WHERE producerCertN = certN
GROUP BY name;
```

Querying Views (IV)

We can also rename attributes within a view definition:

```
CREATE VIEW MovieProd(movieTitle, prodName) AS
   SELECT title, name
   FROM Movie, MovieExec
   WHERE producerCertN = certN;
```

The query

```
SELECT prodName
FROM MovieProd
WHERE movieTitle = 'Gone With the Wind';
```

is transformed by the DBMS into:

```
SELECT E.name
FROM Movie M, MovieExec E
WHERE M.producerCertN = E.certN AND M.title = 'Gone With the Wind';
```

Basic Algorithm for Querying Views

1. Translate the view definition into Relational Algebra;
2. translate the SQL query into Relational Algebra (including the view and all of its operands);
3. substitute the view in the latter algebraic expression by the former algebraic expression and optimize as needed.

View and query definition:

```
CREATE VIEW ParamountMovie AS
  SELECT title, year FROM Movie
  WHERE studioName = 'Paramount';
```

$\pi_{title,\ year}(\sigma_{studioName='Paramount'}(Movie))$

```
SELECT title FROM ParamountMovie
WHERE year = 1979;
```

$\pi_{title}(\sigma_{year=1979}(ParamountMovie))$

Optimized expression:

$$\pi_{title}(\sigma_{year=1979}(\pi_{title,year}(\sigma_{studioName='Paramount'}(Movie)))) \equiv$$

$$\pi_{title}(\sigma_{year=1979\ AND\ studioName='Paramount'}(Movie))$$

Final SQL query:

```
SELECT title FROM Movie WHERE year = 1979 AND studioName = 'Paramount';
```

Modifying Views (I)

Consider the following relation schema.

Movie(title, year, length, inColor, studioName, producerCertN)

Consider the following view definition.

```
CREATE VIEW ParamountMovie2 AS
  SELECT title, year, studioName FROM Movie WHERE studioName = 'Paramount';
```

Simple view modifications are allowed:

```
INSERT INTO ParamountMovie2
   VALUES('Star Trek', 1979, 'Paramount');
```

this is transformed by the DBMS into

```
INSERT INTO Movie
   VALUES('Star Trek', 1979, NULL, NULL, 'Paramount', NULL);
```

```
DELETE FROM ParamountMovie2
  WHERE title LIKE '%Trek%';
```

this is transformed by the DBMS into

```
DELETE FROM Movie
  WHERE title LIKE '%Trek%' AND studioName = 'Paramount';
```

Modifying Views (II)

Consider the following relation schema.

Movie(title, year, length, inColor, studioName, producerCertN)

Consider the following view definition.

```
CREATE VIEW ParamountMovie2 AS
  SELECT title, year, studioName FROM Movie WHERE studioName = 'Paramount';
```

Simple view modifications are allowed:

```
UPDATE ParamountMovie2
SET year = 1979
WHERE title LIKE '%Trek%’ and studioName=‘Fox’;
```

this is transformed by the DBMS into

```
UPDATE Movie
SET year = 1979
WHERE title LIKE '%Trek%' AND studioName = 'Paramount’ and studioName=‘Fox;
```

When are View Modifications Allowed?

Only those views can be modified for which all of the following conditions hold:
SELECT clause has no DISTINCT;
FROM clause contains only a single relation *R*;
WHERE clause does not involve *R* in a subquery;
GROUP BY and HAVING are not used in the view definition;
enough attribute values are specified in the SELECT clause, such that when augmented with NULL values, they become the modified tuple in the base relation.

Recall: NULL values are never allowed under a PRIMARY KEY, but are allowed under a FOREIGN KEY unless when combined with a NOT NULL constraint.

Violations of the above will result in a **compile-time exception**.
Violations of any database constraints will result in a **run-time exception**.

➛ In general, the DBMS has to be able to unambiguously determine the rewriting step required to execute the view update.