Relational Database Systems I

Christoph Lofi
Simon Barthel
Institut für Informationssysteme
Technische Universität Braunschweig
www.ifis.cs.tu-bs.de
• Design quality of database tables can be “improved” by normalization

  – There are several normal forms, which impose restriction to table design

  – Goal of normalization: reduce possibility of update/insert/delete anomalies by reducing redundancy

  – Needed: functional dependencies

    • FD describe dependencies between data values in the real world

      – e.g., a given matriculation number implies a certain student; a city, street name, house number implies a ZIP code
A candidate key of a table is a minimal subset of attributes which (transitively) imply all other attributes

- Primary key: select one of the candidate keys

Normal Forms of a database schema

- 1-NF:
  - Disallows all multi-value attributes

- 2-NF:
  - Is in 1-NF and no non-key attribute is functionally depending on a proper subset of any candidate key
    - Of course, there needs to be a proper subset to violate this constraint (composite candidate keys)
    - Careful: ANY candidate key, not just the chosen primary key
10 Summary

– **3-NF**
  - Is in 2-NF and no key *transitively* determines a non-key attributes
    - Can be achieved by decomposing with respect to the transitive functional dependencies

– **BNCF**
  - Boyce-Codd Normal Form
  - Is in 3-NF and there are no overlapping composite candidate keys
Up to now

– only **direct interaction** with the database via **SQL**

But

– typically, the interaction with the database is **embedded** in some workflow or complex task
– moreover, pure SQL has its limits
  • relationally complete vs. Turing complete
  • it is very hard to express complex operations or data manipulations in pure SQL
    – A **real programming language** would be nice
• Example: **Travel agency**
  
  – **user interaction**
  
  • *I want to go on vacations to Hawai’i in the first week of May.*

  – **basic business workflow**
  
  • check for **flight availability** during the week
  • check for **hotel availability** during the week
  • **align dates** for flights and hotels, shift it around a little for **best prices**
  • **make a reservation** for a suitable hotel room
  • **buy flight ticket** from airline
• External application
  – handles and controls the complete workflow
  – interacts with the database

• Database
  – controls its internal state
    • is the application allowed to access the data?
    • how can data access be sped up?
    • what DML operations are allowed?
• Basically, applications have an external view on the database and simply fetch the data when needed.
Databases have a classical 3-layer architecture

- **application layer**
  - provides interfaces for applications

- **logical layer**
  - contains the representation of the data (data models)
  - controls what happens to the data

- **physical layer**
  - manages the actual storage of the data
    (disk space, access paths, ...)

11.0 Application Programming
• Views
• Indexes
• Transactions
• Accessing databases from applications
  – Embedded SQL
  – SQLJ
11.1 Views

- **Views** provide an external view (i.e., an application’s view) on a database

- **Views** are virtual tables, which (in most respects) can act like physical tables
  - **helps with privacy issues**
    - views may contain only the data a certain user or group is allowed to see
  - **simplifies querying**
    - data is already reduced to the relevant subset
    - data is already aggregated or joined as needed
  - **may increase query evaluation performance**
    - commonly used query expressions can be pre-computed
    - This will induce some performance issues to ensure update consistency
11.1 Views

- **CREATE VIEW** statement

  1. define a name for the view
     - you may use it like a table name later on
  2. optionally, define column names
     - if not, names are taken from the query
  3. optionally, you may specify check options

```
CREATE VIEW view name
(column name)
AS query
WITH CHECK OPTION
```
### 11.1 Views

**Example**

<table>
<thead>
<tr>
<th>mat_no</th>
<th>firstname</th>
<th>lastname</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Clark</td>
<td>Kent</td>
<td>m</td>
</tr>
<tr>
<td>2832</td>
<td>Louise</td>
<td>Lane</td>
<td>f</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
<td>Luther</td>
<td>m</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
<td>Xavier</td>
<td>m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mat_no</th>
<th>crs_no</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>2.0</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
</tr>
<tr>
<td>2832</td>
<td>100</td>
<td>1.3</td>
</tr>
</tbody>
</table>

**results_crs_100**

<table>
<thead>
<tr>
<th>student</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Louise Lane</td>
<td>1.3</td>
</tr>
<tr>
<td>Clark Kent</td>
<td>3.7</td>
</tr>
</tbody>
</table>

**CREATE VIEW** results_crs_100 (student, result) **AS**

**SELECT** (firstname || ' ' || lastname), result

**FROM** exams e, students s

**WHERE** crs_no = 100 AND s.mat_no = e.mat_no
11.1 Views

- Views may also be created without referring to any physical tables
  
  ```
  CREATE VIEW blacklisted_students (firstname, lastname) 
  AS VALUES ('Galan', NULL), ('Norrin', 'Radd')
  ```

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galan</td>
<td>NULL</td>
</tr>
<tr>
<td>Norrin</td>
<td>Radd</td>
</tr>
</tbody>
</table>
11.1 Views

• Generally, views are read-only
  – often, database systems just cannot figure out how to translate view updates into updates of underlying tables

• However, there are updateable views
  – a view is updateable, if its definition does not contain…
    • VALUES, DISTINCT, GROUP BY, HAVING, or column functions
    • any form of joins
    • any reference to a read-only view
    • UNION, INTERSECT, or EXCEPT
      – exception: cleanly partitioned UNION ALL views
11.1 Views

• Examples of the **view update problem**
  – views with projection
    • assume that the primary key from some table has **not** been projected into a view definition
      – project mat_no and result from exams, but not the crs_no
    • any update of the view would have to insert a tuple with primary key NULL into the original table?!
  – views with aggregation
    • assume a view definition computes averages over some groups of tuples
      – take the average grade of each student
    • how can any update of the view be distributed on the original tuples of the table?!
• Depending on the DBMS, the meaning of *updateable* may be different
• Example IBM DB2
  – *deletable*: you may delete rows from the view
  • DB2 needs to be able to map a view row to a single specific (exactly one) row in a single table
  – *updateable*: you may update a given column
  • the view is deletable, and
  • there is a mapping from the column to be updated to exactly one column in the underlying base table
  – *insertable*: you may insert new rows
  • all columns are updateable, and
  • the view definition does not contain UNION ALL
11.1 Views

• Examples

– **CREATE VIEW** statistics **AS**
  SELECT crs_no, AVG(result) **AS** avg_result
  FROM exams **GROUP BY** crs_no
  • Not updatable at all (avg_result is computed)

– **CREATE VIEW** results_crs_100 **AS**
  SELECT firstname, lastname, result
  FROM exams e **JOIN** students s **ON** e.mat_no = s.mat_no
  **WHERE** crs_no = 100
  • not updatable at all
    (each row corresponds to rows across different tables)

– **CREATE VIEW** students_2 **AS**
  SELECT mat_no, firstname, lastname
  FROM students
  • deletable, updatable for each column, and insertable
  • if you insert a new row, the sex will be NULL
11.1 Views: Check Options

• If a view is updateable, you may additionally enforce check options
  – each tuple being inserted or modified needs to match the view definition
  – check-enabled views are called symmetric
    • everything you put into a view can be retrieved from it
    • by default, updateable views are not symmetric
  – two check options
    • local:
      new tuples are only checked within the current view definition
    • cascade (default):
      new tuples are checked recursively within all referenced views
CREATE VIEW results_crs_100 AS
SELECT * FROM exams
WHERE crs_no = 100

CREATE VIEW good_crs_100 AS
SELECT * FROM results_crs_100
WHERE result < 2.7

What happens if you want to insert \( t_1 = (1005, 101, 3.0) \) or \( t_2 = (1005, 101, 2.0) \) into good_crs_100?

- default
  - insert is performed, tuples added to tables but not visible in any view
- LOCAL CHECK OPTION on good_crs_100
  - \( t_1 \) cannot be added, \( t_2 \) can be added but is not visible
- CASCADE CHECK OPTION on good_crs_100
  - \( t_1 \) cannot be added, \( t_2 \) cannot be added
11.1 Views: Materialization

• In SQL-92, views were intended to be a mechanism for **query rewriting**
  – views were just a shortcut, queries containing views were changed by the DBMS in more complex queries containing the view definition
  – view is re-evaluated every time it is used!

• However, some DBMS allow to **materialize** views
  – may drastically **increase performance**
  – view is **physically created and updated** when the dependent tables change
  – useful, if query creating the view is very time-consuming, data very stable, and storage space is not an issue
    • Usually, useful when \#reads>>\#writes
• In DB2, materialized views are called **materialized query tables (MQTs)**
  – use `CREATE TABLE` statement like a view definition
  – always **read-only**
  – specify additional table update policies

```sql
CREATE TABLE view name (column name) AS
  query
  DATA INITIALLY DEFERRED
  REFRESH IMMEDIATE
  REFRESH DEFERRED
```
11.1 Views: Materialization

• By default, the table is filled with the query results
  – **DATA INITIALLY DEFERRED** does not fill the table automatically, but creates an empty one

• You may choose when the table is updated
  – automatically (**REFRESH IMMEDIATE**): table is updated whenever the contents of one of the underlying tables changes
  – manually (**REFRESH DEFERRED**): you must manually update the table
  • Use **REFRESH TABLE table_name**
• Views
• Indexes
• Transactions
• Accessing databases from applications
  – Embedded SQL
  – SQLJ
11.2 Indexes

• **Indexes** are used to speed up database retrieval
  – basically an index is a special **access path** to the data
  – the data is **ordered** with respect to one (or more) attribute(s) according to the index
  – think: Encyclopedia Britannica
    • when looking for a term, you do not scan over all 32 volumes
11.2 Indexes

• Indexes...
  – can influence the **actual storage** of the data for sequential reading in table scans
  – or can just be an ordered collection of **pointers** to the data items

• Search time is **massively reduced**
  – typical index structures are B-trees, R*-trees or bitmap indexes

• All **details** in Relational Database Systems 2 (next semester)
11.2 Indexes

• Typically, we have two types of indexes:
  
  – **Primary Index:**
    • Created by default for the primary key attributes of a table
    • Index physically reorders the whole table
      – Think: Ordering of topics in an encyclopedia by alphabet
    • Efficient search is possible
      – Forward search, skip-forward search, binary search, etc.
  
  – **Secondary Index:**
    • Optional indexes for non-primary key attributes
    • Extremely beneficial for speeding up **joins** on foreign key constraints!
    • Builds an additional data structure containing the index
      – Usually, this is a B-Tree
      – Costs space for storage and time for updates
11.2 Indexes

• DB admins can create **many indexes** on a table, but the number of indexes should be limited
  
  – each index carries a certain **cost**!
  
  • part of the cost is paid in **space**, since some data is replicated
  
  • part of the cost is paid in **update performance**, since each update has to be reflected in all indexes including the column
  
  – what indexes to chose mainly depends on the query load (**physical database tuning**).
11.2 Indexes

- Create or delete an index over some (list of) attribute(s) as follows:

\[
\text{CREATE INDEX } \text{index name} \text{ ON table name (column name \{ASC, DESC\})}
\]

\[
\text{DROP INDEX } \text{index name}
\]
11.2 Indexes

• **Primary key columns** have an index by default
• Also for each UNIQUE constraint, there is a corresponding index by default
• Certain restrictions may apply for index creation
  – e.g., in **IBM DB2**
    • an index can include at most 16 attributes
    • other constraints are imposed by table space properties (physical storage)
11.2 Indexes

- After creating indexes, **statistical information** should be collected to help the DB optimizer making best use of the new index.
- Also, many DBMS offer **system-specific** options during index creation.
  - physical index type, possible scan directions, index update behavior, ...
What indexes you need to create heavily depends on your application
- part of physical DB tuning
- physical DB tuning is a complicated and non-transparent task

Usually done **heuristically** by **trial-and-error**

1. **identify** performance problems
2. **measure** some hopefully meaningful performance metrics
   - based on common queries or queries creating problems
3. **adjust** the current index design
   - create new indexes with different properties
4. **measure** again
   - if result is better: Great! Continue tuning (if needed)!
   - if result is worse: Bad! **Undo** everything you did and try something else.
11.2 Indexes: Examples

- Example database: IMDb data
  - Internet Movie Database
  - contains (among other data)
    - 1,181,300 movies of 7 types
    - 2,226,551 persons
    - 15,387,808 associations between actors and movies
11.2 Indexes: Examples

- Create indexes for example query
  - *Which cinema movies before 1986 featured Harrison Ford?*
11.2 Indexes: Examples

• SQL query
  – SELECT t.title, t.production_year
    FROM title t JOIN cast_info c ON (t.id = c.movie_id)
    JOIN name n ON (c.person_id = n.id)
    JOIN kind_type k ON (t.kind_id = k.id)
    WHERE n.name = 'Ford, Harrison',
    AND n.imdb_index = 'I',
    AND t.production_year < 1986
    AND k.kind = 'movie'

• Execution statistics without index
  – ~ 283 000 time units (around 30 seconds…)

Relational Database Systems 1 – Christoph Lofi – Institut für Informationssysteme – TU Braunschweig
11.2 Indexes: Examples

• Indexes help reducing search times on attributes
• **Analyze query:** Which searches are performed?
  – c.person_id = n.id
  – c.movie_id = t.id
  – n.name = 'Ford, Harrison'
  – t.production_year < 1986
  – ...
• **Create indexes** for the columns involved in selections and joins
  – actually, this is a very coarse heuristic
  – in reality, you would use **EXPLAIN** statements to identify needed indexes (or an automatic *index advisor*)
    • see our lecture Relational Database Systems 2
11.2 Indexes: Examples

• Simple index creation

  – CREATE INDEX title_year
      ON title (production_year)
  – CREATE INDEX name_name
      ON name (name)
  – CREATE INDEX cast_info_person
      ON cast_info (person_id)
  – CREATE INDEX cast_info_movie
      ON cast_info (movie_id)
  – ...

Relational Database Systems 1 – Christoph Lofi – Institut für Informationssysteme – TU Braunschweig
• After indexes have been created, query evaluates faster, even by several orders of magnitudes
  – 71 time units (instant response) compared to 283 000 time units (~30 seconds)
  – performance increased by 4000% !!!
• Views
• Indexes
• Transactions
• Accessing databases from applications
  – Embedded SQL
  – SQLJ
11.3 Transactions

• Sometimes operations on a database depend on each other

  – example: money transfers in banking applications
    • deducing the amount from one account and adding it on
      another should always happen together
    • if only one part happens the database is incorrect and money
      vanishes, which is bad

  – such connected operations are bundled by the underlying
    workflows
11.3 Transactions

- Workflows require the concept of **transactions**
  - a transaction is a **finite set of operations** that have to be performed in a certain **sequence**, while ensuring **recoverability** and certain **properties**

- These properties are concerned with
  - **integrity**: transactions can always be executed safely, especially in concurrent manner, while ensuring data integrity
  - **fail safety/recovery**: transactions are immune to system failures
The properties that ensure the transactional properties of a workflow are known as the **ACID principle**

- Atomicity
- Consistency
- Isolation
- Durability

Every system handling **non-ACID transactions** has to take special precautions.
11.3 Transactions: ACID

- **Atomicity**
  - any transaction is either executed completely or not at all

- **Consistency (preservation)**
  - transactions lead from one consistent state of the data instance to another

- **Isolation**
  - transactions are isolated from others, i.e., even in a concurrent scenario transactions do not interfere with each other

- **Durability**
  - as soon as the transaction is completed (committed), all data changes performed are guaranteed to survive subsequent system failures
Transactions

- **SQL** supports transactions
  - a transaction is *implicitly started* on the first access to the database
  - any sequence of operations performed by some application can either be *ended* with...
    - a **COMMIT** statement (also **COMMIT WORK**) successfully closing the transaction and saving all changed data persistently to the database
    - a **ROLLBACK** statement (also **ROLLBACK WORK**) aborting the transaction and leaving the database in the same state it was in before starting the transaction
    - a transaction can be divided into several steps by setting so-called **savepoints**: then rollbacks can also be performed **partially** step-by-step, one savepoint at a time
11.3 Transactions

• When interacting with databases
  – whenever the database is in auto-commit mode, each single SQL statement is considered a transaction
    • a COMMIT is automatically performed after the execution of each statement
    • if the statement was a query, a COMMIT is automatically performed after the result set has been closed
  – the COMMIT or ROLLBACK command has to be explicitly stated
11.3 Transactions

UPDATE hero
  SET name = 'Jean Grey-Summers'
  WHERE name = 'Jean Grey'
UPDATE hero
  SET name = 'Scott Grey-Summers'
  WHERE name = 'Scott Summers'
COMMIT;

DELETE FROM alias WHERE hero_id = 1;
DELETE FROM hero WHERE id = 1;
SAVEPOINT deleted1;
DELETE FROM alias WHERE hero_id = 2;
DELETE FROM hero WHERE id = 2;
ROLLBACK TO deleted1;
COMMIT;

Auto-Commit must be disabled!
• Views
• Indexes
• Transactions
• **Accessing databases from applications**
  – Embedded SQL
  – SQLJ

Relational Database Systems 1 – Christoph Lofi – Institut für Informationssysteme – TU Braunschweig
11.4 Accessing Databases

• Applications are usually programmed in some high-level language
  – C, C++, C#, Java, Perl, PHP, Cobol, etc.

• Main problems
  – how does an application connect to a DBMS?
  – how are queries (SQL) integrated into the application’s programming language?
  – how are result sets handled and converted into the language’s data formats?
  – how are advanced DBMS features accessed from within the programming language?
11.4 Accessing Databases

**Application Layer**

- ATM
- Travel agency
- ... bookkeeper
- Applications

**DBMS Layer**

- Encapsulated data
- Exposed data
- View
- DBMS
- DB pages

- Applications clients
There are three major approaches

1. **directly embed** all database commands into the host language
   - oldest approach
   - examples
     - EmbeddedSQL for C
     - SQLJ for Java

2. design a specialized **DB programming language**
   - rarely used
   - example
     - Oracle PL/SQL
3. using a **library (API)** to connect to the database

- most popular approach
  - chances are good that you will use it in the future…

- major examples
  - **CLI** (call level interface)
  - **ODBC** (Open Database Connectivity)
  - **JDBC** (Java Database Connectivity)

- covered in the next lecture
When dealing with programming languages and databases, a common problem is the **impedance mismatch**

- programming language and database use different **data models**
  - how to map between them?
- **DB: relational model**
  - tables with rows and columns
  - attributes with their data types
- **host language**
  - different data types, often no explicit NULL values
  - usually no native support for table structures compatible with DBs
  - different data models
    - **object-oriented data models**
    - **record-oriented data models**
• SQL statements are **embedded** directly into the host language
  – This is not a popular approach anymore
  – Currently, the trend is towards using APIs (e.g., JDBC) or stored procedures

• A **precompiler** parses the host code, extracts and compiles the SQL statements
  – the SQL statements are compiled by the **host compiler** into a native DB calls
  – host SQL is replaced by calls to the compiled SQL

• **Compiler** transforms host language into executables
11.4 Embedded SQL in C

- **Example:** EmbeddedSQL for C

```
program.pc
```

- **SQL precompiler**

```
program.c
```

- **C compiler**

```
program.o
```

- **Linker**

```
program
```

- **C source code with embedded SQL**

```
SQL host compiler
```

- **Native C with calls to compiled SQL**

```
Binary object code
```

- **SQL libraries**

```
Executable DB program
```
11.4 Embedded SQL in C

- SQL statements are usually started by the keyword `EXEC SQL` and terminated by `;` or `END EXEC`.
- To bridge between SQL and C, shared variables are used.
  - Constants in SQL can be replaced by C variables prefixed by `:` in SQL.
  - Shared variables carry their current value into SQL.
  - Shared variables explicitly declared in the `declare section`.
- When a `SELECT` statement is used, it is followed by the `INTO clause` listing the shared variables holding the query result.
Return a single SQL result

```
// usual C code goes here
EXEC SQL BEGIN DECLARE SECTION;
  int matNo;
  int avgResult;
EXEC SQL END DECLARE SECTION;
// usual C code goes here, e.g. set value for matNo
EXEC SQL
  SELECT avg(result) INTO :avgResult
  FROM exams e WHERE e.matNr = :matNo
END EXEC
// usual C code goes here, e.g. do something with avgResult
```
What happens if the query result is not a scalar value but a result table?

– when multiple rows are returned, a **row cursor** is used to access them sequentially
– fights the **SQL/C impedance mismatch**

A cursor provides a **pointer to a single row** in the query result (which may have many rows)

– cursors are **declared by statements similar to view definitions** (declaration includes a query)
One result row at a time is accessed through the cursor, which is moved to the next row before each data transfer. The columns of that one row are ‘fetched’ into the program variables which can then be manipulated in the normal way by the host program. A cursor can also be used to update values in tables.
To work with cursors, you need the following commands

- **DECLARE**: defines the cursor with its query
- **OPEN**: executes query, creates result set on DB side and resets cursor (before the first row)
- **WHENEVER NOT FOUND**: guards the next SQL statement. If anything goes wrong, a given action is performed.
- **FETCH**: increments the cursor and returns values of next row. Failure if there is no next row.
- **CLOSE**: closes cursor and frees server-side resources
11.4 Embedded SQL in C

Return a SQL multi-set result

```c
// ...
EXEC SQL BEGIN DECLARE SECTION;
    int matNo;
    int result;
EXEC SQL END DECLARE SECTION;
// set matNo here
EXEC SQL DECLARE result_cursor CURSOR FOR
    SELECT result FROM exams WHERE matNr = :matNo;
EXEC SQL OPEN result_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO done;
while (1) {
    EXEC SQL FETCH result_cursor INTO :result;
    // do something with the result using C
}
done: ;
EXEC SQL CLOSE result_cursor;
// ...
```
11.4 SQLJ

• Embedding SQL directly was also adopted by Java as **SQLJ**
  
  – **SQLJ** is newer than the more popular **JDBC**, but conceptually very close to embedded SQL
    • ISO standard
    • also uses precompiler
    • syntax is shorter than JDBC
  
  – **SQLJ** is recommended for **user defined functions (UDF)** in newer versions of Oracle and DB2
    • UDFs are covered in the next lecture
  
  – statements are **precompiled**
    • syntax can be checked during **compile time**
    • queries can be **pre-optimized** and thus may provide superior performance compared to true dynamic JDBC SQL
      – however, most SQLJ implementations just convert SQLJ to JDBC calls without any optimization
• SQL statements are in-fixed by `#sql{...}`
• SQL statements may use any Java variable as *shared* variables
  – in contrast to embedded SQL, no explicit declaration in SQLJ is necessary
  – also prefixed by : 
• Failing SQL statements fire normal Java *exceptions*
• For retrieving results with several columns and rows, (non-standard) *iterators* are declared
Return a single SQL result

```java
int matNo = 12;
int averageResult;
try {
    try {
        \#sql{SELECT avg(result) INTO :averageResult FROM exams e WHERE e.matNr = :matNo}
    } catch (SQLException ex) {
        // ...
    }
```
Return an SQL multi-set result

```java
try {
    #sql iterator CrsStat(int crsNo, double avgResult);
    CrsStat stat = null;
    #sql stat = {SELECT crsNr, avg(result)
                  FROM exams GROUP BY crsNr}
    while (stat.next()) {
        // do something with stat.crsNo & stat.avgResult
    }
} catch (SQLException ex) {
    // ...
}
```
Today’s techniques are very very important in database applications!
- Learn when and how to use them!

Views
- Views create logical “tables” from a given query
  - Can be used, for example, for easier querying or for security reasons
- Can be materialized
  - May speed up some things, but also introduces issues with consistency
- Are usually read-only, but can be updatable
  - Only if certain requirements are fulfilled
**Indexes**

- Used to locate tuples quicker
  - Usually, will lead to faster queries
- **Primary indexes**
  - Imply the physical order of tuples in the table with respect to the primary key attributes
- **Secondary indexes**
  - Optional indexes for non-primary key attributes which create additional data structures
  - Beneficial for foreign key attributes if join queries are often used
Transactions

- Groups multiple database SQL statements into one functional unit
  - Default in many DBMS: Each single statement is a transaction
- Important for **critical applications** and **multi-user environments**
  - Prevents anomalies and inconsistencies resulting from, e.g., database failures and concurrent accesses
- Each group of statements is executed in a **controlled** fashion
  - **ACID** properties: Atomicity, Consistency, Isolation, Durability
  - “Transactions are either executed fully or not at all, and will not be affected by partial results of other database transactions”
• Database APIs
  – CLI
  – ODBC
  – JDBC