• 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
11.0 Application Programming

• External application
  – handles and controls the complete workflow
  – interacts with the database

• Database
  – controls the 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

<table>
<thead>
<tr>
<th>exams</th>
<th>crsNr</th>
<th>matNr</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1002</td>
<td>1.7</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>1002</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>1005</td>
<td>1.7</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>2832</td>
<td>3.7</td>
<td></td>
</tr>
</tbody>
</table>
• Databases have a 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
  – ODBC
  – JDBC
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

```sql
CREATE VIEW view_name
(column_name)
AS query
WITH [CASCADED | LOCAL]
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>

**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` (first_name, last_name) `AS VALUES` (‘Galan’, NULL), (‘Norrin’, ‘Radd’)

<table>
<thead>
<tr>
<th>first_name</th>
<th>last_name</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?!
11.1 Views

- Depending on the DBMS, the meaning of \textit{updateable} may be different
- Example IBM DB2
  - \textbf{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 (usually via primary key)
  - \textbf{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
  - \textbf{insertable}: you may insert new rows
    - all columns are updateable, and
    - the view definition does not contain UNION ALL
    - Does not violate \texttt{not null} constraints
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
    - If not, you can insert tuples in the underlying table which won’t appear in the view
  - 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
11.1 Views: Check Options

- 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
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 read performance.
- view is physically created and updated when the dependent tables change.
  - …this, of course, decreases write performance.
- useful, if query creating the view is very time-consuming, data very stable, and storage space is not an issue.
  - Usually, useful when \#reads\gg\gg\gg\gg\gg\#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

```
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
  – ODBC
  – JDBC
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)
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 choose 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:

```sql
CREATE INDEX index_name ON table_name (column_name [ASC | DESC]),
```

**Index creation:****
- `CREATE INDEX` - Creates an index.
- `UNIQUE` - Ensures uniqueness of the index.

**Index deletion:**
- `DROP INDEX` - Deletes an index.
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, ...

Relational Database Systems 1 – Wolf-Tilo Balke – Institut für Informationssysteme – TU Braunschweig
• 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**
  
  ```sql
  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 – Wolf-Tilo Balke – 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

Relational Database Systems 1 – Wolf-Tilo Balke – Institut für Informationssysteme – TU Braunschweig 33
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)
  – ...
11.2 Indexes: Examples

• 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% !!!
11 Application Programming I

- Views
- Indexes
- Transactions
- Accessing databases from applications
  - ODBC
  - JDBC
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
11.3 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

```sql
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 for this to work!
11 Application Programming I

- Views
- Indexes
- Transactions
- **Accessing databases from applications**
  - ODBC
  - JDBC
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

DBMS layer

ATM

travel agency

bookkeeper

applications clients

applications

DBMS

DB pages

view

encapsulated data

exposed data

app₁

... appₙ
11.4 Accessing Databases

• There are four major approaches

• **directly embed** all database commands into the host language
  
  – oldest approach
  
  • EmbeddedSQL for C
  
  • SQLJ for Java

• design a specialized **DB programming language**
  
  – rarely used
  
  – Example: Oracle PL/SQL
• using low-level SQL library (API) to connect to the database
  – most popular approach
    • our focus in this lecture
    • 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)
• using a high-level ORM (object relational mapper) library to transparently translate between DB and programming language
  – Our focus of the next lecture
11.4 Accessing Databases

• 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**
• General steps in using database APIs
  – set up the environment
  – define and establish connections to the DBMS
    • This will create the server-side resources for user interaction
  – create and execute statements
    • Results create a result object on the DB server
  – process the results
    • using the cursor concept, iteratively transfer result tuples
  – close the connections
    • Frees all resources again
11.4 ODBC

• The **Open Database Connectivity (ODBC)** provides a standardized API to DBMS
  – development driven by **Microsoft** in 1992, later versions aligned with X/Open and ISO/IEC
  – builds on **several CLI specifications**, but does not implement full SQL features
  – central for the design was **independence** of programming language, operating system, and DBMS
  – implements the standardized **middleware** concept
11.4 ODBC

- **Basic idea:** The DBMS is virtualized
  - Application programmer does not need **specialized knowledge** about the chosen DBMS
    - This is all handled by the driver which maps generic DBMS calls to vendor-specific commands
  - Drivers handled by operating system
Being a **middleware solution** a basic implementation of ODBC always contains…

- a generic **ODBC driver manager library** to interpret the applications’ commands
  - defines standard types and features
- and a set of **database drivers** to provide the DBMS-specific details
  - each database vendors can write an individual driver to map ODBC commands
11.4 JDBC

• **JDBC** provides a standard **Java** library for accessing tabular data
  
  – tabular data usually means a **relational DBMS**
  
  – API provides standard way to **connect** to a DB
  
  – API allows to perform **dynamic queries**
  
  – method to create stored (parameterized) queries
  
  – provides some (limited) data types for Java/DB **impedance mismatch**
    
    • result sets with rows and columns (cursor concept)
    
    • methods for accessing table **meta data**
  
  – provides functionality **independent** of chosen DBMS
11.4 JDBC

- JDBC does not abstract from SQL statements
  - SQL statements are treated as **Java strings**
  - in case of full dynamic SQL, sometimes excessive **string manipulation** is necessary
  - if DBMS uses different/extended SQL syntax, this has to be considered by the programmer

- JDBC is not an acronym, but a registered product **trademark** by Oracle (used to be Sun Micros.)
  - however, usually, it is assumed that it stands for **Java Database Connectivity**
11.4 JDBC

• Why not just use ODBC?
  – ODBC is based on binary libraries (usually written in C)
    • native calls necessary
    • not platform-independent which is one of Java’s goals
    • ODBC drivers are often registered in the OS
  – 1:1 translation from ODBC to Java does not work as ODBC heavily relies on pointers
  – ODBC API is more complex and littered (and thus harder to learn and use)
    • for example, programmer needs to worry about byte alignment and advanced connection properties explicitly
  – intention was to create a “pure” Java and fully portable API
    • no installation required, JDBC can easily be bundled into the application archive
11.4 JDBC

• JDBC is composed of two primary components

• JDBC API: An programming interface for database connectivity.
  – written in 100% pure Java
  – completely independent of platform, vendor, and DBMS
  – provided by the Oracle in its Java SDK by default
    • usually to be found in java.sql and javax.sql
11.4 JDBC

• JDBC driver
  – implementation of the respective API interface, responsible for communication with the database
  – interface implementation in Java, but may call binary libraries, middleware, or other tools
  – heavily dependent on the used DBMS
  – usually provided by the DB vendor
11.4 JDBC

• General Architecture
  – java application uses API
  – API uses driver
  – driver communicates with DB

• If you change the DBMS, you need to
  – provide a new driver
  – change configuration of driver
  – assuming the SQL syntax is compatible, you are done
    • if not, you are in trouble…
11.4 JDBC: Versions

- There are several versions of JDBC, each with improved functionality

<table>
<thead>
<tr>
<th>Version</th>
<th>Year</th>
<th>Java Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC 4.2</td>
<td>2013</td>
<td>Java 8</td>
</tr>
<tr>
<td>JDBC 4.1</td>
<td>2011</td>
<td>Java 7</td>
</tr>
<tr>
<td>JDBC 4.0</td>
<td>2006</td>
<td>Java 6</td>
</tr>
<tr>
<td>JDBC 3.0</td>
<td>2001</td>
<td>Java 1.4 &amp; Java 5</td>
</tr>
<tr>
<td>JDBC 2.1</td>
<td>1999</td>
<td>Java 1.3</td>
</tr>
<tr>
<td>JDBC 1.2</td>
<td>1997</td>
<td>Java 1.1</td>
</tr>
</tbody>
</table>

- JDBC drivers are written for a specific JDBC version
  - driver should match the JDBC version
  - however, most features also work with outdated drivers

- JDBC 3.0 is still very common!
  - However, 4.0 provides many major and valuable improvements!
11.4 JDBC: Levels

• Beside versions, there are JDBC levels
  – comparable to ODBC tiers
  – for each level, there are different drivers
    • be careful when picking your driver! You need the right version and correct level!
  – all levels offer the same functionality (i.e., API is the same), but use different means of driver implementation and communication with the DBMS
    • different performance and portability properties
11.4 JDBC: Levels

• **Level 1: JDBC/ODBC bridge**
  
  – JDBC driver just translates requests to ODBC calls
    
    • **performance overhead** due to translation
  
  – needs correctly installed ODBC drivers on every client machine
    
    • distribution difficult
    
    • ODBC drivers are not platform-independent
11.4 JDBC: Levels

- **Level 2: Native API**
  - JDBC driver uses native calls to connect to a proprietary client software which handles DB connections
    - e.g. ORACLE client (which is a 1.7 GB installation)
  - difficult to port and with deployment problems
  - often used as cheap-and-dirty solution for older systems
    - also, may be a good idea if application is running on the same machine as the DBMS
11.4 JDBC: Levels

• **Level 3: Middleware**
  
  – JDBC driver communicates with a **middleware** software instead of the DBMS
  
  – often used for **large-scale enterprise applications** in a multi-tier-environment
  
  – vendor specific translation may happen at the middleware
    • just one client driver for any used DBMS
  
  – middleware encapsulates the actual DBMS
    • useful for advanced clustering, extended security, logging, caching, pooling, etc..
### 11.4 JDBC - Levels

**Level 4: Direct pure Java**

- Driver purely written in Java
  - No call translation
  - No installation, no deployment problems
  - Full portability due to platform-independence

- Driver connects directly to the DBMS
  - You need a different driver for each different DBMS
  - Superior performance in remote scenarios
  - For access to a local DBMS, Level 1 might be better
11.4 JDBC

• Basic steps when working with JDBC

1. **Load** the driver
2. **Define** a connection URL
3. **Establish** a connection
4. **Create** a statement(s)
5. **Execute** a statement(s)
6. **Process** the result(s)
7. **Close** the connection
11.4 JDBC: Create a Connection

• The **connection** is necessary to interact with a DBMS Server
  – Established by a **Driver** provided by your DBMS vendor
  – The DBMS server is specified using a **URL**
    • `jdbc:[driverAlias]:[driverParameters]`
    • DB2 Level 4: URL
      – `jdbc:db2://[server][:port]/[db-name]`
      – example: `jdbc:db2://myserver.de:50000/hero_db`
    • SQLite Level 4 URL:
      – `jdbc:sqlite:[filename]`
      – example: `jdbc:sqlite:test.db`
11.4 JDBC: Create a Connection

• The **Driver** instances are normally not managed manually
  – Instead use the **DriverManager**
    • `DriverManager.getConnection(String URL)` provides you with a `Connection` object
    • Automatically searches a suitable driver in the Classpath matching the URL
      – Driver implements an `acceptsURL(String URL)` method
    • In earlier JDBC Versions it was necessary to load and register Drivers using
      – `Class.forName("path.to.driver");`

```java
public Connection getConnection(Properties props) throws SQLException {
    return DriverManager.getConnection("jdbc:db2://dblab.ifis.cs.tu-bs.de:50000/DBLAB", props);
}
```
To actually execute an SQL statement you need a `Statement` object

- Created by a `Connection` object
- There are three types of Statements
  - `Statement`
    - SQL directly written in the Statement
    - To be executed once
  - `PreparedStatement`
    - Used for frequent statements
    - Statement is provided as parameterized String
    - For each execution, parameters are replaced by values
  - `CallableStatement`
    - Used to execute server-side stored procedure (UDF)
11.4 JDBC: Statements

• Using simple **Statement** objects
  – create a **Statement object** with the connection
    • `conn.createStatement();`
  – call one of the three execution methods
    • `executeQuery(String query):`
      – use for **SELECT** queries
      – returns a **ResultSet**
    • `executeUpdate(String query):`
      – use for any DDL or data changing statements (**INSERT, UPDATE, DELETE**)
      – returns an integer with number of affected rows
    • `execute(String query):`
      – advanced feature for multi-**ResultSet** queries

```java
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT count(*) FROM IMDB.title");
```
To access a query result, JDBC provides a `ResultSet`:
- rows are retrieved one after another from the server
  - inspired by (but not compatible to) the Java iterators
  - a `cursor` is held pointing the current row in the server-side result set
- at first, the result set points `before` the first row
  - so, initially to no row at all
- `next()` method moves the cursor to the next row
  - returns true, if there is a next row

```java
ResultSet rs = stmt.executeQuery("SELECT ...")
while(rs.next()) {
    // do something with the current row
}
```
To read the columns of a row, there are multiple getters
- named `getX()` (e.g. `getInt()`, `getDouble()`)
  - access columns by name or by number (starting at 1)
- there are getters for each data type
  - each SQL data type is mapped to a Java data type

```java
ResultSet rs = stmt.executeQuery(
    "SELECT id, real_name FROM heroes"
);
while(rs.next()) {
    int id = rs.getInt(1);
    String realName = rs.getString("real_name");
    System.out.println(id + ":" + realName);
}
```
## 11.4 JDBC: Data Types

**Example: Extract from direct JDBC data types for DB2**

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>SQL Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, java.lang.Integer</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long, java.lang.Long</td>
<td>BIGINT</td>
</tr>
<tr>
<td>double, java.lang.Double</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CLOB(n)</td>
</tr>
<tr>
<td>java.io.StringReader</td>
<td>CLOB(n)</td>
</tr>
<tr>
<td>byte[]</td>
<td>BLOB(n)</td>
</tr>
<tr>
<td>java.io.ByteArrayInputStream</td>
<td>BLOB(n)</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>DATE</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>TIME</td>
</tr>
</tbody>
</table>
• After the result set rows has been read, the statement is marked **complete**
  – Statements and Results are usually **garbage collected** by Java
  – however, it is highly recommended to **manually close** statements (and thus result sets)
    • `stmt.close()`
    • potentially saves system resources
11.4 JDBC: Result Set Options

• Default ResultSet only allows moving the cursor forward and is read-only
  – can be manipulated by using Navigation options:
    • TYPE_FORWARD_ONLY (default) to allow only forward movement
    • TYPE_SCROLL_INSENSITIVE to allow forward, backward and random movement
      – previous(): moves cursor to the previous row
      – beforeFirst(): moves cursor before the first row
      – relative(int x): moves cursor x rows forward (or backward if x is negative)
      – absolute(int x): moves cursor to the given absolute row number
11.4 JDBC: Result Set Options

- **TYPE_SCROLL_SENSITIVE**
  - same as **TYPE_SCROLL_INSENSITIVE**
  - **but** changes to the underlying data are directly visible in the result (i.e. rows are always up-to-date)
  - may have bad performance

```java
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_READ_ONLY
);
```
11.4 JDBC: Result Set Options

• **Update options**
  
  – **CONCUR_READ_ONLY** (default)
    
    • results can only be read
    
    • unlimited concurrency
  
  – **CONCUR_UPDATABLE** to allow updates
    
    • use `updateX` methods to update the current row (similar to `getX` methods)
      
      – updates are performed after you call `updateRow()`
      
      – if you want to cancel the updates, call `cancelRowUpdates()`
      
      – if you move to another row without `updateRow()`, nothing happens
  
  • may **degenerate performance** in massively concurrent applications due to lock contention thrashing (see RDB2)
• **CONCUR_UPDATABLE** (continued)
  
  – can also be used to **insert rows**
    
    • use `moveToInsertRow()` to move to a special **insert row**
    • use `update-`methods to set the values for the new row
    • then call `insertRow()` to commit the insert (cursor returns to the previous position)
  
  – or to **delete rows**
    
    • call `deleteRow()` to delete the current row

```java
Statement stmt = conn.createStatement(
    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_UPDATABLE
);
ResultSet rs = stmt.executeQuery("SELECT id, real_name FROM heroes");
while (rs.next()){
    rs.updateString("real_name",
        rs.getString("real_name").toUpperCase());
    rs.updateRow();
}
```
11.4 JDBC: Result Set Options

```java
Statement stmt = conn.createStatement(
    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_UPDATABLE
);
ResultSet rs = stmt.executeQuery(
    "SELECT id, real_name FROM heroes"
);

rs.moveToInsertRow();
rs.updateInt(1, 999);
rs.updateString(2, "Peter Parker");
rs.insertRow();

while (rs.next()) {
    if (rs.getString("real_name").equals("Bruce Banner")) {
        rs.deleteRow();
    }
}
```
11.4 JDBC: Receive Metadata

- The database **metadata** can be accessed via a **DatabaseMetaData** object
  - can be received using the connection
    - using `conn.getMetaData();`
  - Metadata includes
    - DBMS name, version, installation properties
    - available **schemas**, **tables**, **columns**
    - **primary keys** for a given table
    - ...

```java
DatabaseMetaData metaData = conn.getMetaData();
String dbmsName = metaData.getDatabaseProductName();
ResultSet schemas = metaData.getSchemas();
```
• Programming Languages and DBMS #2
• Active Databases
• Basic Security