



ifis

Institut für Informationssysteme
Technische Universität Braunschweig

Relational Database Systems I

Wolf-Tilo Balke

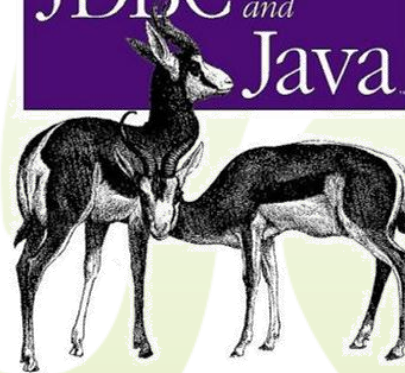
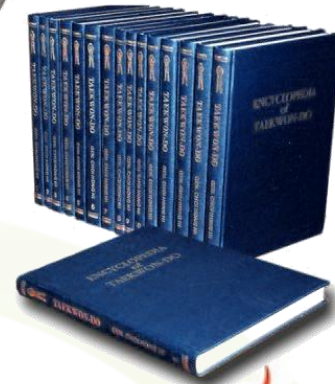
Simon Barthel

Institut für Informationssysteme
Technische Universität Braunschweig
www.ifis.cs.tu-bs.de



Overview

- Advanced **database concepts** for application programming
 - **Views**
 - **Indexes**
 - **Transactions**
- **Accessing databases** from applications
 - Embedded SQL
 - SQLJ





II Application Programming

- 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



II Application Programming

- Example: **Travel agency**

- **User interaction**

- “I want to go on vacations to Hawaii 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





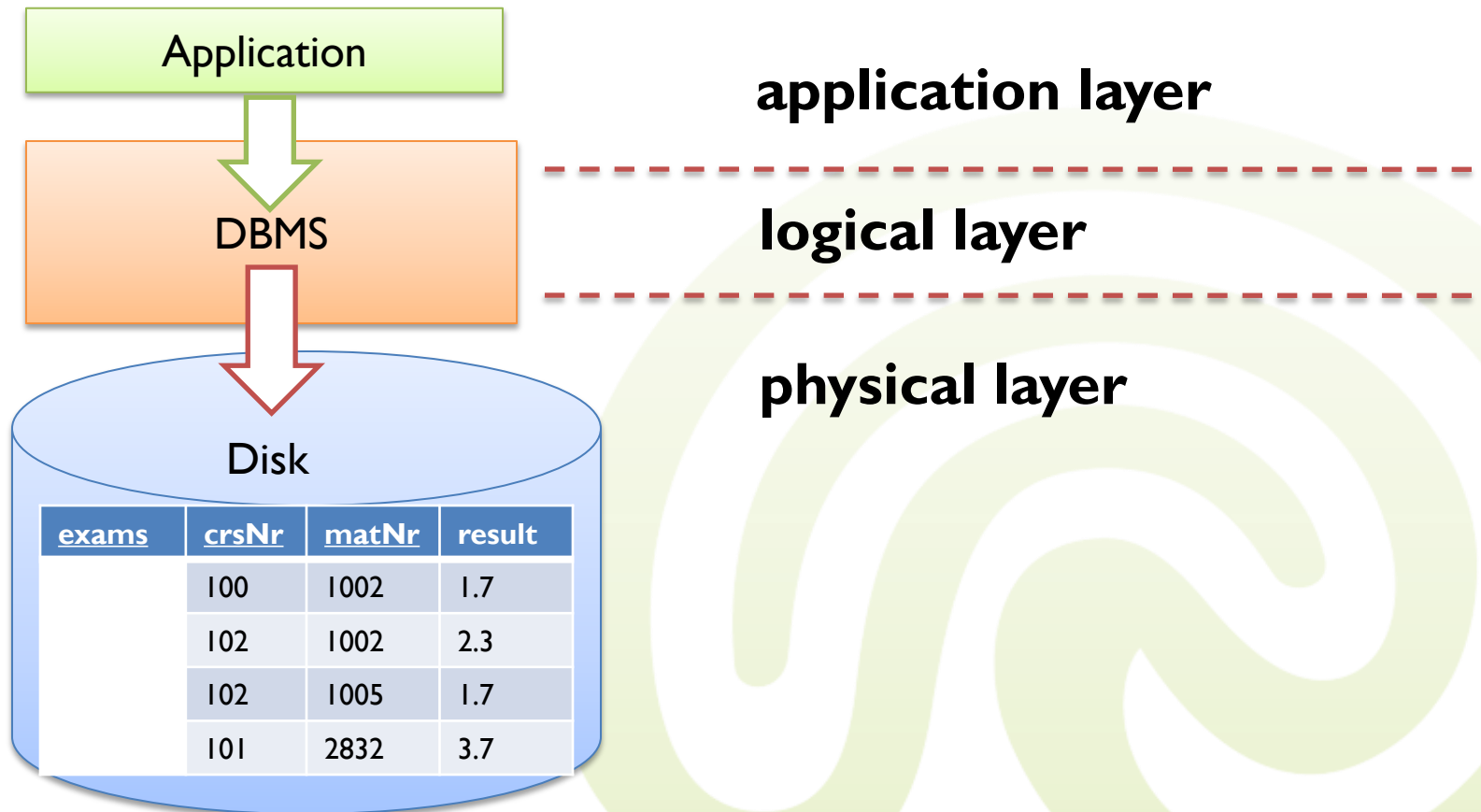
II Application Programming

- **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?



II Application Programming

- Basically, applications have an external view on the database and simply fetch the data when needed





II Application Programming

- 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, ...)



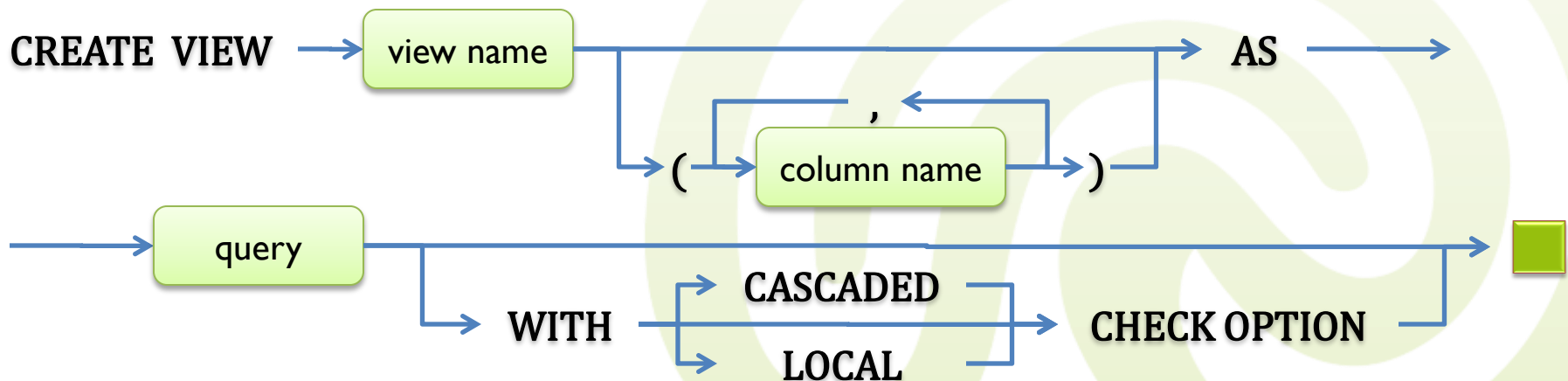
11.1 Views

- **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 precomputed



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





11.1 Views

- Example:

students

matNr	firstName	lastName	sex
1005	Clark	Kent	m
2832	Louise	Lane	f
4512	Lex	Luther	m
5119	Charles	Xavier	m

exams

matNr	crsNr	result
1005	100	3.7
2832	102	2.0
1005	101	4.0
2832	100	1.3

resultsCrs100

student	result
Louise Lane	1.3
Clark Kent	3.7

```
CREATE VIEW resultsCrs100 (student, result) AS
SELECT (firstName || ' ' || lastName), result
FROM exams e, students s
WHERE crsNr = 100 AND s.matNr = e.matNr
```



11.1 Views

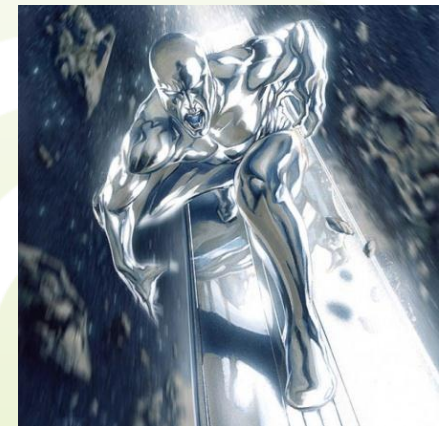
- Views may also be created without referring to any physical tables

- **CREATE VIEW**

blacklistedStudents (firstName, lastName)
AS VALUES ('Galan', NULL), ('Norrrin', 'Radd')

blacklistedStudents

firstName	lastName
Galan	NULL
Norrrin	Radd





11.1 Views

- Generally, views are **read-only**
 - 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 matNr and result from exams, but not the crsNr
 - 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 “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 crsNr, AVG(result) **AS** avgResult
FROM exams **GROUP BY** crsNr
 - Not updatable at all (avgResult is computed)
 - **CREATE VIEW** resultsCrs100 **AS**
SELECT firstName, lastName, result
FROM exams e **JOIN** students s **ON** e.matNr = s.matNr
WHERE crsNr = 100
 - Not updatable at all
(each row corresponds to rows across different tables)
 - **CREATE VIEW** students2 **AS**
SELECT matrNr, 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



11.1 Views: Check Options

- **CREATE VIEW** resultsCrs100 **AS**
 SELECT * **FROM** exams
 WHERE crsNr = 100
- **CREATE VIEW** goodCrs100 **AS**
 SELECT * **FROM** resultsCrs100
 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 goodCrs100?
 - **Default**
 - Insert is performed, tuples added to tables but not visible in any view
 - **LOCAL CHECK OPTION** on goodCrs100
 - t_1 cannot be added, t_2 can be added but is not visible
 - **CASCADE CHECK OPTION** on goodCrs100
 - t_1 cannot be added, t_2 cannot be added




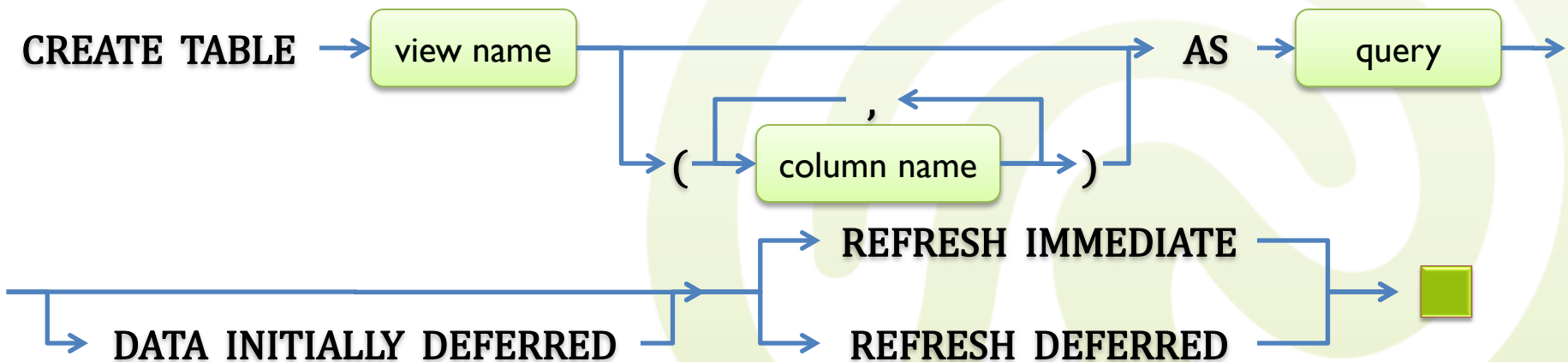
11.1 Views: Materialization

- In SQL-92, views were intended to be a mechanism for **query rewriting**
 - View 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



11.1 Views: Materialization

- 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





11.1 Views: Materialization

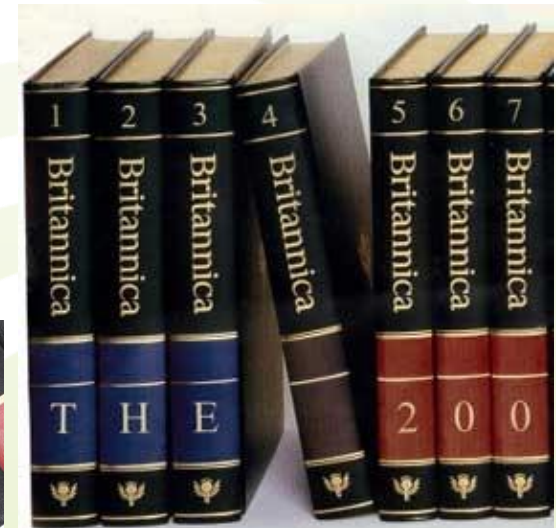
- By default, the table is filled with the query results
 - **DATA INITIALLY DEFERRED** does not fill the table automatically, but creates an empty one
- You may choose when the table is updated
 - **Automatically (REFRESH IMMEDIATE):**
Table is updated whenever the contents of one of the underlying tables changes
 - **Manually (REFRESH DEFERRED):**
You must manually update the table
 - Use `REFRESH TABLE tableName`





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

- 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**)

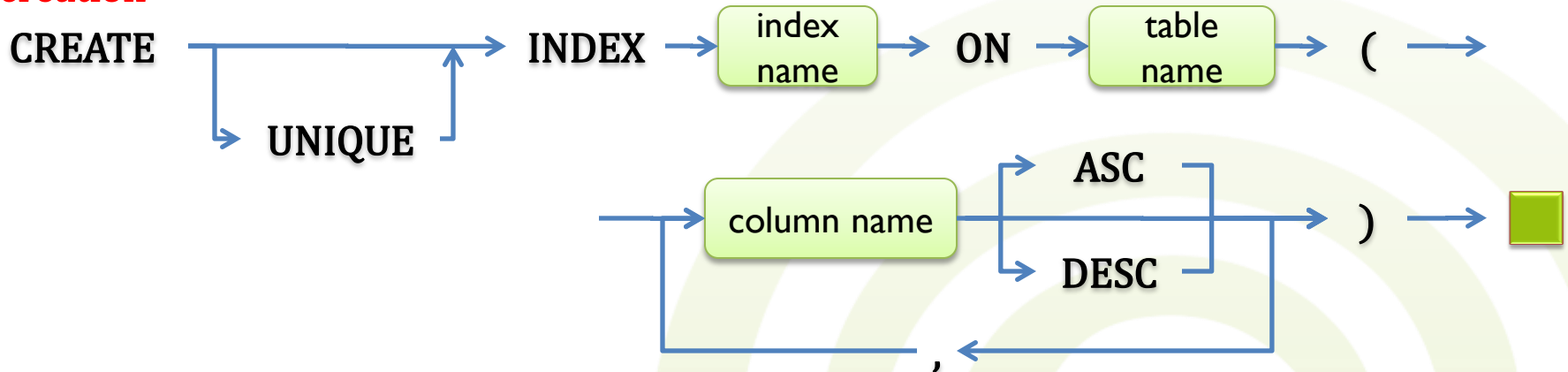




11.2 Indexes

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

index
creation



index
deletion





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
 - For example, 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, ...



11.2 Indexes: Examples

Detour

- 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.

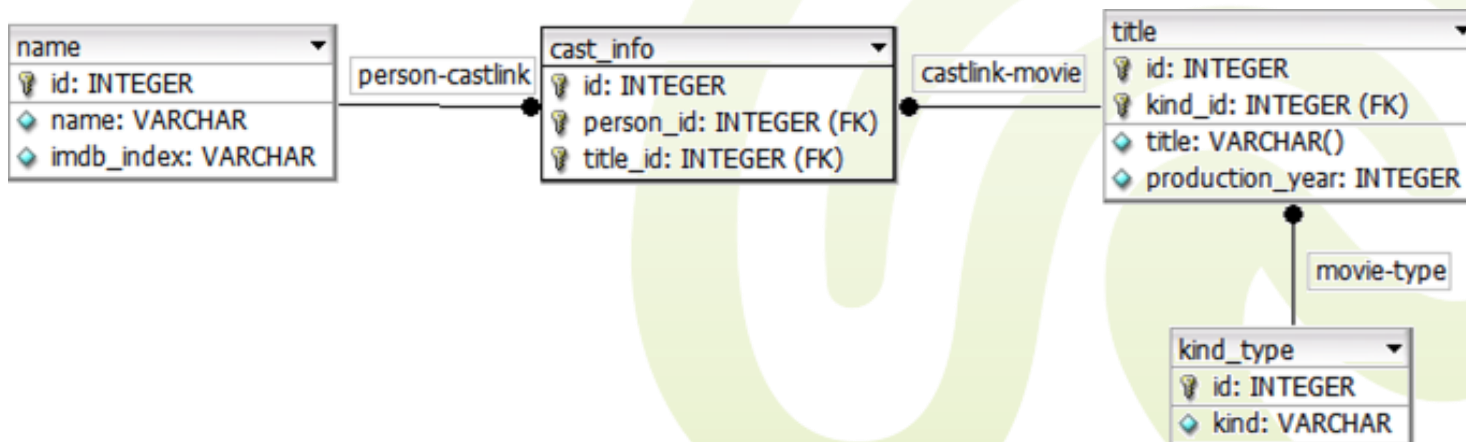




11.2 Indexes: Examples

Detour

- 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

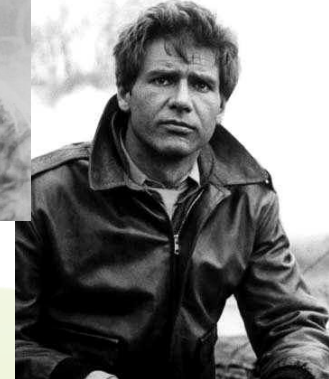
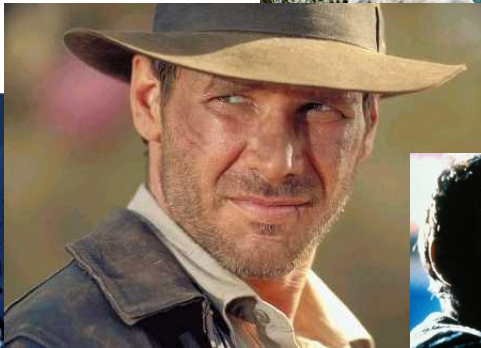




11.2 Indexes: Examples

Detour

- Create indexes for example query
 - “Which cinema movies before 1986 featured Harrison Ford?”





- **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...)



11.2 Indexes: Examples

Detour

- 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

Detour

- 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

Detour

- 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% !!!**

TITLE	PRODUCTION_YEAR
Dead Heat on a Merry-Go-Round	1966
Time for Killing, A	1967
Luv	1967
Journey to Shiloh	1968
Zabriskie Point	1970
Getting Straight	1970
American Graffiti	1973
Conversation, The	1974
Star Wars	1977
Heroes	1977
Force 10 from Navarone	1978
More American Graffiti	1979
Hanover Street	1979
Frisco Kid, The	1979
Apocalypse Now	1979
Star Wars: Episode V - The Empire Strikes Back	1980
Raiders of the Lost Ark	1981
Blade Runner	1982
Star Wars: Episode VI - Return of the Jedi	1983
Indiana Jones and the Temple of Doom	1984
Witness	1985



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



11.3 Transactions: ACID

- The properties that ensure the transactional properties of a workflow are known as the **ACID principle**
 - **A**tomicity
 - **C**onsistency
 - **I**solation
 - **D**urability
- 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

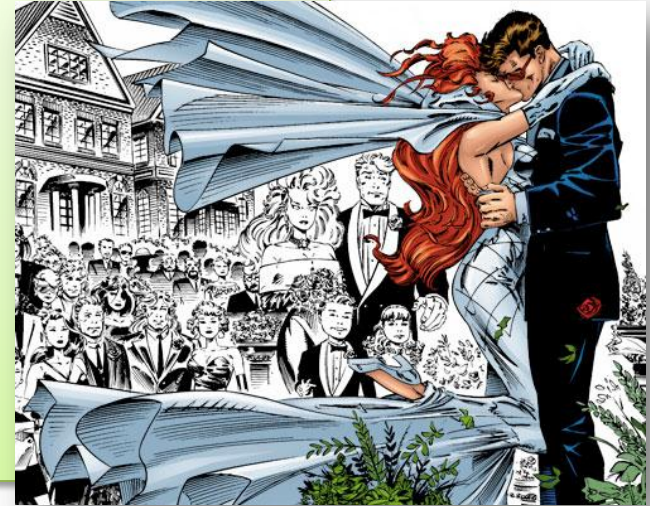




11.3 Transactions

```
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!

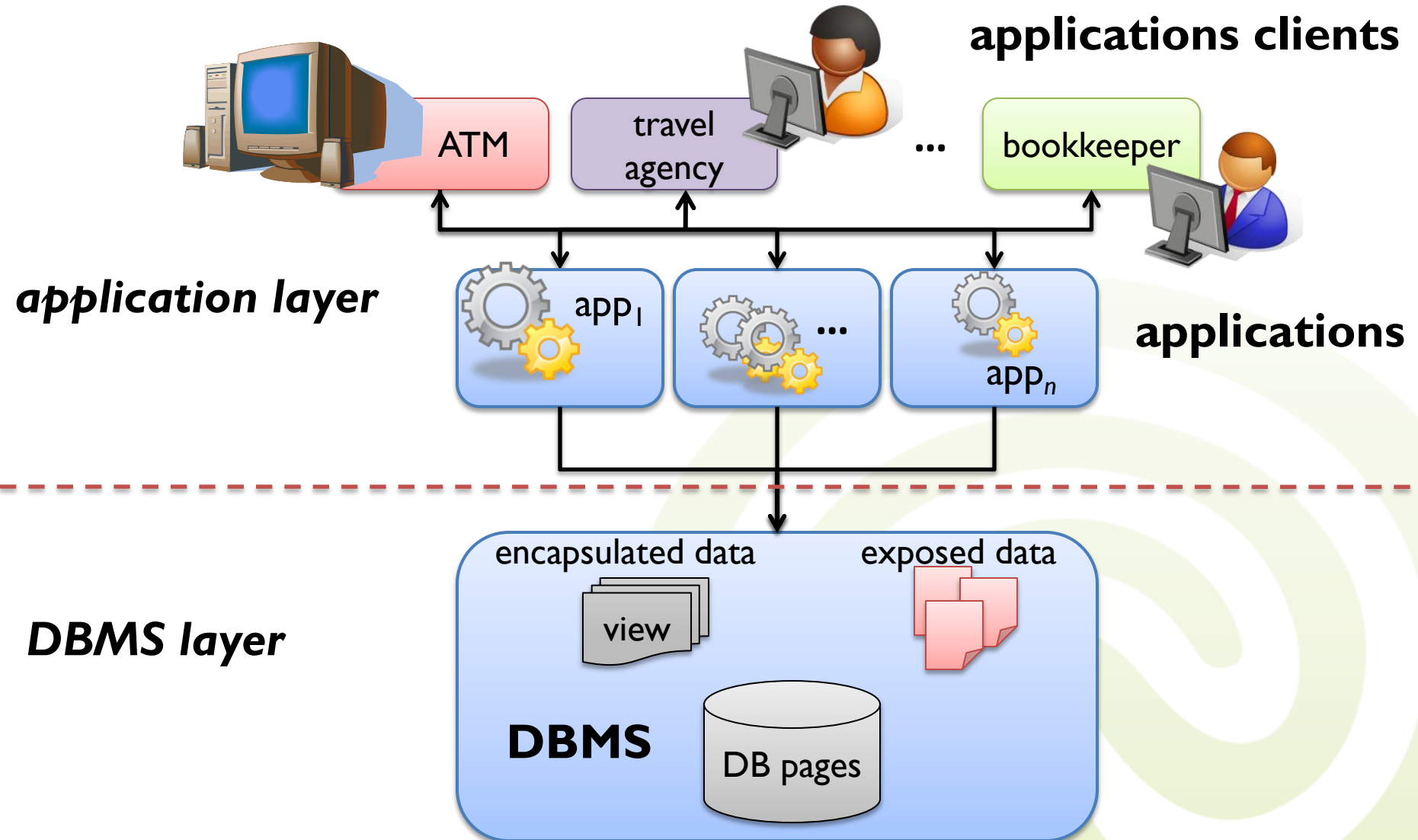


11.4 Accessing Databases

- 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





11.4 Accessing Databases

- 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



11.4 Accessing Databases

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



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**

JAVA != SQL



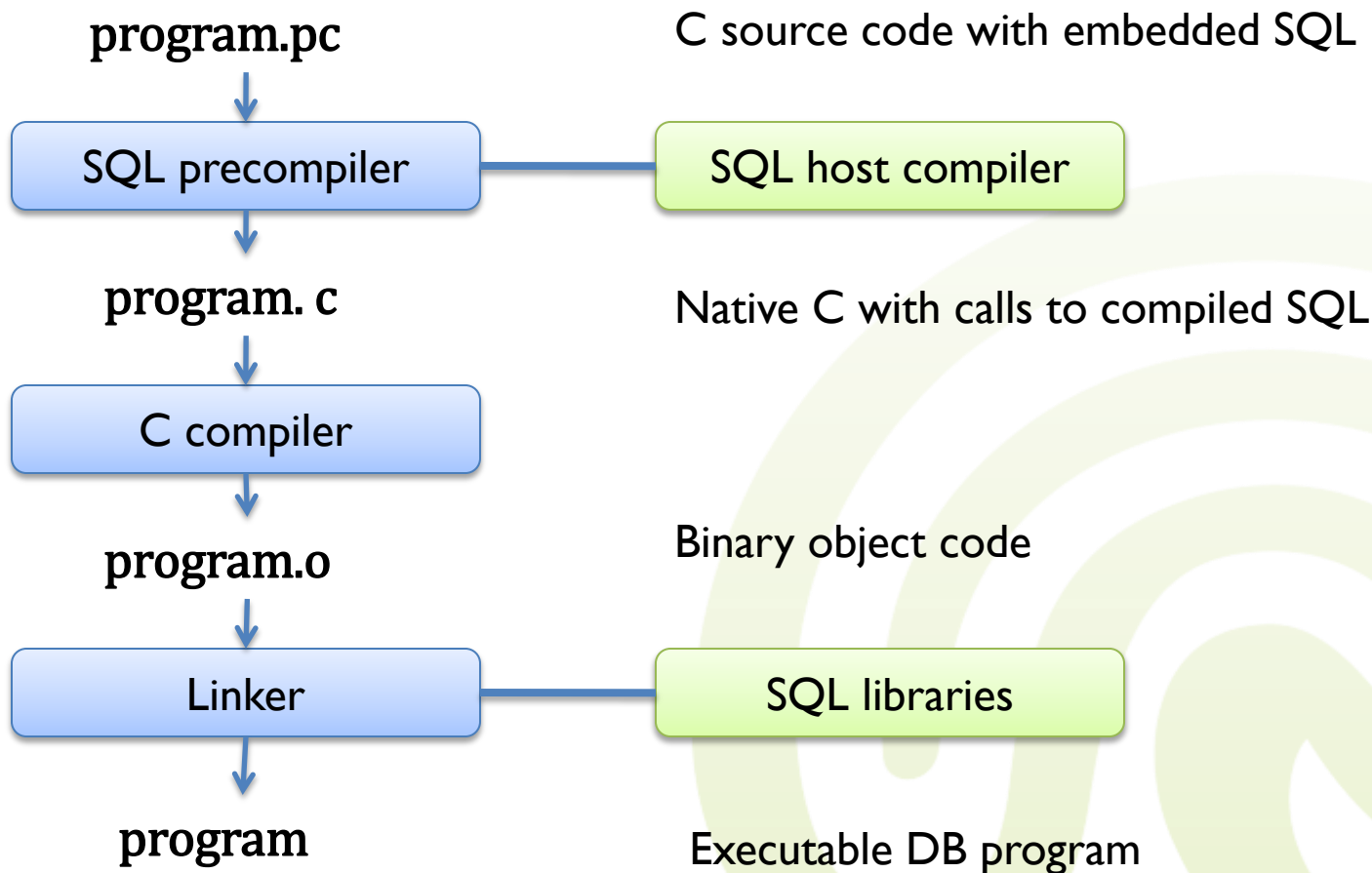
11.5 Embedded SQL

Detour

- SQL statements are **embedded** directly into the host language
- 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



- Example: **EmbeddedSQL for 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 pre-fixed 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



11.5 Embedded SQL in C

Detour

Return a single SQL result

```
// usual C code goes here
EXEC SQL BEGIN DECLARE SECTION;
    int matNr;
    int avgResult;
EXEC SQL END DECLARE SECTION;
// usual C code goes here, e.g. set value for matNr
EXEC SQL
    SELECT avg(result) INTO :avgResult
    FROM exams e WHERE e.matNr = :matNr
END EXEC
// usual C code goes here, e.g. do something with avgResult
```



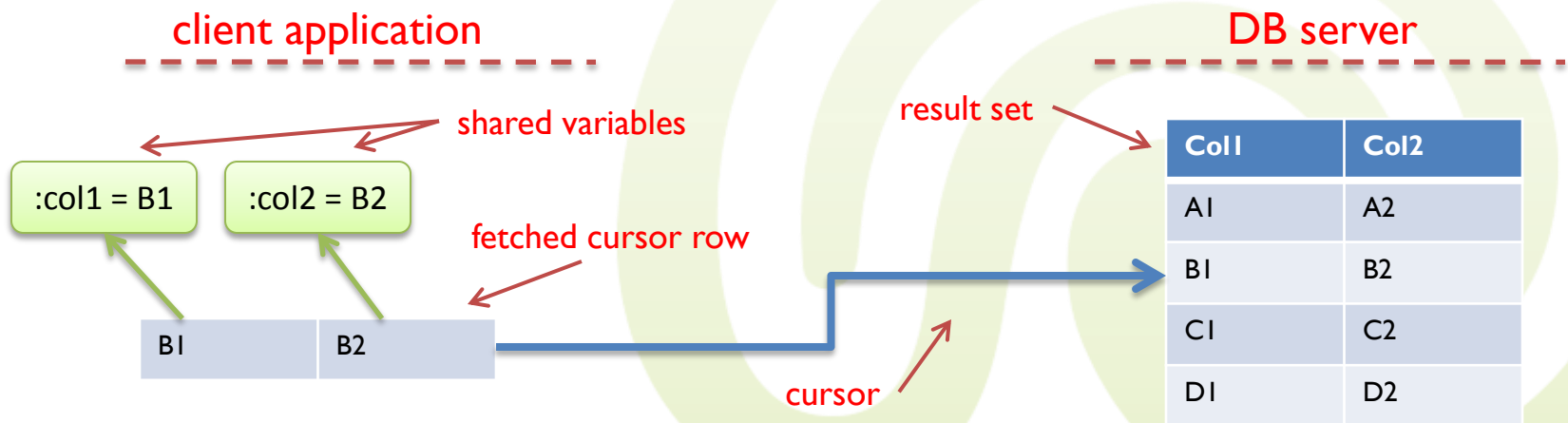
- What happens if the query result is not a scalar value but a result table?
 - When multiple rows are returned, a **row cursor** is used to access them sequentially
 - Fights the **SQL/C impedance mismatch**
- A cursor provides a **pointer to a single row** in the query result (which may have many rows)
 - Cursors are **declared by statements** similar to **view definitions** (declaration includes a query)



11.5 Embedded SQL in C

Detour

- **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



11.5 Embedded SQL in C

Detour

Return a SQL multi-set result

```
// ...  
EXEC SQL BEGIN DECLARE SECTION;  
    int matNr;  
    int result;  
EXEC SQL END DECLARE SECTION;  
// set matNr here  
EXEC SQL DECLARE result_cursor CURSOR FOR  
    SELECT result FROM exams WHERE matNr = :matNr ;  
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;  
// ...
```



11.6 SQLJ

Detour

- 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

```
int matNr = 12 ;  
int averageResult;  
try {  
    #sql{SELECT avg(result) INTO :averageResult  
          FROM exams e WHERE e.matNr = :matNr}  
} catch (SQLException ex) {  
    // ...  
}
```




Return an SQL multi-set result

```
try {  
    #sql iterator CrsStat(int crsNr, double avgResult);  
    CrsStat stat = null;  
    #sql stat = {SELECT crsNr, avg(result)  
                FROM exams GROUP BY crsNr}  
    while (stat.next()) {  
        // do something with stat.crsNr & stat.avgResult  
    }  
} catch (SQLException ex) {  
    // ...  
}
```



Next Lecture

- **Database APIs**
 - CLI
 - ODBC
 - JDBC

