Database example

- Library Database

**Database example**

- Customers(`Cust_Id`, `Cust_Name`, Faculty)
  - `Cust_Id`: Customer ID (unique)
  - `Cust_Name`: Customer Name
  - `Faculty`: Faculty Name

**Database example**

- Books(`Book_Id`, `Book_Name`, Year, Max_Time, Faculty, Pages)
  - `Book_Id`: Unique book id
  - `Book_Name`: Book title
  - `Year`: Year of print
  - `Max_Time`: Maximum borrowing time in days
  - `Faculty`: Faculty name
  - `Pages`: Page number

---

**Database example**

- Customers(`Cust_Id`, `Cust_Name`, Faculty)
Database example

- **Ordered(Cust_Id, Book_Id, Order_Date)**
  - **Cust_Id**: Customer ID
  - **Book_Id**: Book ID
  - **Order_Date**: Date of book order

<table>
<thead>
<tr>
<th>Cust_Id</th>
<th>Book_Id</th>
<th>Order_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>1111</td>
<td>14-Oct-2002</td>
</tr>
<tr>
<td>45678</td>
<td>1112</td>
<td>24-Oct-2002</td>
</tr>
<tr>
<td>12345</td>
<td>1113</td>
<td>30-Oct-2002</td>
</tr>
<tr>
<td>45678</td>
<td>2222</td>
<td>12-Oct-2002</td>
</tr>
</tbody>
</table>

Database example

- **Borrowed(Book_Id, Cust_Id, From_Date, To_Date)**
  - **Book_Id**: Book ID
  - **Cust_Id**: Customer ID
  - **From_Date**: Borrowing date
  - **To_Date**: Return date

<table>
<thead>
<tr>
<th>Book_Id</th>
<th>Cust_Id</th>
<th>From_Date</th>
<th>To_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>5555</td>
<td>56789</td>
<td>13-Oct-2002</td>
<td></td>
</tr>
</tbody>
</table>

**Outline**

- Sorting
- LIMIT and OFFSET
- Set Operations
- WITH
- TRIGGERS

**SQL Queries: Sort**

```
SELECT select_list
FROM table_expression
ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
[, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```
SQL Queries: Sort

- Return the customers names and ids
  - ordered by
    - customer name (in ascending order) and by
    - ids in descending order

```sql
SELECT cust_name, cust_id
FROM customers
ORDER BY cust_name, cust_id DESC
```

SQL Queries: Sort

- Customers(Cust_Id, Cust_Name, Faculty)

<table>
<thead>
<tr>
<th>Cust_Id</th>
<th>Cust_Name</th>
<th>Faculty</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>Moshe Cohen</td>
<td>CS</td>
</tr>
<tr>
<td>23456</td>
<td>Avi Barak</td>
<td>EE</td>
</tr>
<tr>
<td>34567</td>
<td>Avi Barak</td>
<td>MED</td>
</tr>
<tr>
<td>45678</td>
<td>Lior Edri</td>
<td>EE</td>
</tr>
<tr>
<td>56789</td>
<td>Moshe Cohen</td>
<td>EE</td>
</tr>
<tr>
<td>67890</td>
<td>Moshe Cohen</td>
<td>EE</td>
</tr>
</tbody>
</table>

SQL Queries: LIMIT & OFFSET

- Must use ORDER BY while using LIMIT/OFFSET
- Allows to return a range of sorted result
- The indices are determined by the ORDER BY

```sql
SELECT cust_name, cust_id
FROM customers
ORDER BY cust_name, cust_id DESC
OFFSET 2
LIMIT 2
```

SQL Queries: LIMIT & OFFSET

- Return the 3rd and 4th customers from previous query

```sql
SELECT cust_name, cust_id
FROM customers
ORDER BY cust_name, cust_id DESC
```
SQL Queries: SET Operations

- **UNION** query1 UNION [ALL] query2
- **INTERSECT** query1 INTERSECT [ALL] query2
- **EXCEPT** query1 EXCEPT [ALL] query2

SQL Queries: SET Operations

- Recall
  - A table is a bag of tuples
  - Not a set since the same tuple can appear several times
- We can specify how to handle duplicates:
  - To keep them we write the operation name + ALL
  - To eliminate we just write the operation name
- To perform set operations schemas must be **type-equal**

Example: Return the customers id that preformed an action in the library (borrowed or ordered a book)

```sql
SELECT Cust_Id from orderd
UNION
SELECT Cust_Id from borrowed
```

```
<table>
<thead>
<tr>
<th>Cust_Id</th>
<th>Book_Id</th>
<th>Order_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>1111</td>
<td>14-Oct-2002</td>
</tr>
<tr>
<td>4678</td>
<td>1112</td>
<td>24-Oct-2002</td>
</tr>
<tr>
<td>12345</td>
<td>1113</td>
<td>30-Oct-2002</td>
</tr>
<tr>
<td>4678</td>
<td>2222</td>
<td>12-Oct-2002</td>
</tr>
</tbody>
</table>
```

```sql
SELECT Cust_Id from orderd
UNION
SELECT Cust_Id from borrowed
```

```
<table>
<thead>
<tr>
<th>Cust_Id</th>
<th>Book_Id</th>
<th>From_Date</th>
<th>To_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>5555</td>
<td>56789</td>
<td>13-Oct-2002</td>
<td></td>
</tr>
</tbody>
</table>
```

```sql
SELECT Cust_Id from orderd
UNION
SELECT Cust_Id from borrowed
```

```
<table>
<thead>
<tr>
<th>Cust_Id</th>
<th>Book_Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>56789</td>
<td></td>
</tr>
</tbody>
</table>
```
SQL Queries: SET Operations

```
SELECT Cust_Id from orderd
UNION
SELECT from_date from borrowed
```

Problem – date and ID are of different types

---

SQL Queries: WITH

- WITH creates a temporary table to use within the query you write
- Unlike views, this temporary table is calculated only as a part of the query, and its definition is not saved on the server.

```
WITH TempCSBooks as (
  SELECT Book_Id, Book_Name
  FROM Books
  WHERE Faculty = 'CS'
)
SELECT * FROM TempCSBooks
```

---

SQL Queries: WITH RECURSIVE

- WITH RECURSIVE allows to perform calculations that cannot be done with standard SQL syntax
- Calculations are done until reaches fixpoint
  - The current calculations does not change the result of the previous one
- Example: calculate the sum of all the numbers from 1 to 100

```
WITH RECURSIVE t(n) AS (
  VALUES (1)
  UNION
  SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

---

SQL Queries: WITH RECURSIVE

- Given a directed graph as relation of edges (source, destination), find all paths that do not include a cycle
- This cannot be done without recursion
  - For general graphs, the lengths are not bounded
  - Thus, we need to know in advance how much JOIN operations to make

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

---

SQL Queries: WITH RECURSIVE

- To recognize whether there is a cycle, we have to save all of the nodes in the path.
- For that we use arrays:
  - Arrays:
    - Add element \ array concatenation : ||
      - \[1,2,3\] || 4 \=> \[1,2,3,4\]
    - You can use the ANY \ ALL operator to check if a value is in array
      - 4 =ANY \[1,2,3,4\]
  - More information:
SQL Queries: WITH RECURSIVE

WITH RECURSIVE find_paths(source, destination, length, path) AS (
    SELECT source, destination, 1, ARRAY[source]
    FROM edges e
    UNION ALL
    SELECT fp.source, e.destination, fp.length + 1, path || e.source
    FROM edges e, find_paths fp
    WHERE e.source = fp.destination AND NOT (e.destination = ANY(path))
)
SELECT * FROM find_paths
order by source, destination, length;

<table>
<thead>
<tr>
<th>source</th>
<th>destination</th>
<th>length</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>[1]</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>[1]</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>[1, 2]</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>[1, 2]</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>[1]</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>[1, 2]</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>[1, 2, 3]</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>[1]</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>[2]</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>[3]</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>[1, 4]</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>[1]</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>[1, 2]</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>[4]</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>[4, 2]</td>
</tr>
</tbody>
</table>

1/24/2018