Databases	Lecture 5
ENGR 3599	Feb 21, 2019

Structured Query Language

Relational algebra describes operations we can perform on relations that we can use to answer queries. It leads to a very procedural query language: you have to describe the sequence of operations to apply in detail.

A declarative query language only requires you to describe what you want to query, not how.

SQL (Structured Query Language) has become the standard for declarative query languages over relational data. It is one of the reasons for the success of relational database systems. It has been standardized, and you can think of it as a common API for relational databases.

Examples will be based on the following relations:

Account:

account	type	balance
991	CH	1000
992	SA	500
993	CH	50
994	CH	100
995	CD	10000

Client:

ssn	name
1111	R
2222	T
3333	A

HasAccount:

account	ssn
991	1111
992	1111
993	2222
994	1111
994	3333
995	1111

Basic queries: A basic SQL query looks as follows:

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualifications
```

where *target-list* is a list of attribute names,[1] *relation-list* is a list of table names, and *qualifications* is a list of conditions on tuples, conditions involving comparisons of the form *attribute op attribute* or *attribute op constant* where *op* is a comparison operator such as `=`, `>`, `<`, `>=`, `<=`, `<>`, and combined using `AND`, `OR`, and `NOT`. The `DISTINCT` qualifier indicates that the ruples in the result should not contain any duplicates.

Conceptual evaluation strategy:

1. Compute product of tables in *relation-list*
2. Discard tuples that fail *qualifications*
3. Discard attributes not in *target-list*
4. If `DISTINCT`, eliminate duplicate tuples

With `DISTINCT` this roughly translates to the relational algebra expression:

$$\pi_{\textit{target-list}}(\sigma_{\textit{qualifications}}(r_1 \times \cdots \times r_k))$$

where $r_1, \ldots, r_k$ are the relations in *relation-list*. (Without `DISTINCT`, we need a version of relational algebra with multisets, as discussed last week.)

The above conceptual evaluation strategy is just that, conceptual. It is not the most efficient way of evaluating queries, but it establishes a reference.

Example:

```
SELECT C.name
FROM Client C, HasAccount H
WHERE C.ssn = H.ssn AND H.account = 994
```

yielding the relation:

C.name
R
A

[1] The special symbol `*` means *all attributes*.

```
SELECT DISTINCT H.ssn
FROM HasAcount H, Account A
WHERE A.account = H.account AND A.type = 'CH'
```

yielding the relation:

H.ssn
1111
2222
3333

Nested queries and set conditions: Here is another query to return the name of clients with account 994, this time using a nested query (which behaves like a set):

```
SELECT C.name
FROM Client C
WHERE C.ssn IN (SELECT H.ssn
                FROM HasAccount H
                WHERE H.account = 994)
```

Conceptually, this checks the qualifications of each tuple by computing the subquery, and checking whether `C.ssn` appears in the result column of the subquery (which is required to be a one-column relation, whose column name is irrelevant). Note that in this case the subquery can be computed once and its result used for every tuple.

```
SELECT C.name
FROM Client C
WHERE EXISTS (SELECT *
              FROM HasAccount H
              WHERE H.ssn = C.ssn and H.account = 994)
```

Conceptually, this checks whether the relation described by the subquery is non-empty. We can check that this computed the same result as above, although note that now the subquery must be explicitly re-evaluated for every tuple of `Client`, since it depends on `C.ssn`.

The following query uses `IN` in a way that is more difficult to avoid (unlike the above examples). It returns the clients with both a checking and a savings account:

```
SELECT H.ssn
```

```
FROM HasAccount H, Account A
WHERE H.account = A.account AND A.type = 'CH'
  AND H.ssn IN (SELECT H2.ssn
                FROM HasAccount H2, Account A2
                WHERE H2.account = A2.account AND A2.type = 'SA')
```

Aggregation: Lets you aggregate values of attributes across tuples.

Types of aggregation:

SUM ([DISTINCT] *attr*) AS *name*
AVG ([DISTINCT] *attr*) AS *name*
MAX (*attr*) AS *name*
MAX (*attr*) AS *name*
COUNT ([DISTINCT] *attr*) AS *name*

Examples:

```
SELECT COUNT(C.ssn) AS count
FROM Client C
```

yielding:

count
3

```
SELECT SUM(A.balance) AS total
FROM Account A
```

yielding:

total
11650

```
SELECT AVG(A.balance) AS avg_checking
FROM Account A
WHERE A.type = 'CH'
```

yielding:

avg_checking
383.33

```
SELECT SUM(A.balance) AS total_1111
FROM HasAccount H, Account A
WHERE H.account = A.account AND H.ssn = 1111
```

yielding:

total_1111
2900

Aggregation with grouping: Lets you aggregate values of attributes across tuples, partitioned into groups defined by the value of a set of grouping attributes.

Example:

```
SELECT H.ssn, SUM(A.balance) as total
FROM HasAccount H, Account A
WHERE H.account = A.account
GROUP BY H.ssn
```

yielding:

H.ssn	total
1111	11600
2222	50
3333	100

Conceptual evaluation strategy: The general form of a query with aggregation and grouping is:

SELECT *target-list*
FROM *relation-list*
WHERE *qualifications*
GROUP BY *grouping-list*
HAVING *group-qualifications*

1. Compute product of tables in *relation-list*
2. Discard tuples that fail *qualifications*
3. Discard attributes not in *target-list*
4. Partition into groups by values of attributes in *grouping-list*
5. Discard groups that fail *group-qualifications*
6. Aggregate tuples in each group

Example: Maximum balance of clients with more than one non-CD account.

```
SELECT H.ssn, MAX(A.balance) as max
FROM HasAccount H, Account A
WHERE A.type <> 'CD'
GROUP BY H.ssn
HAVING COUNT(A.balance) > 1
```

SQL and CRUD operations We can think of a `SELECT` query as a read operation on the tuples in relations. What about the other CRUD operations? SQL provides ways to create, update, and delete tuples from relations:

```
INSERT INTO table
VALUES (value,...,value)
```

```
UPDATE table
SET attribute = value,
    ...,
    attribute = value
WHERE qualifications
```

```
DELETE FROM table
WHERE qualifications
```

Database systems also provide a Data Definition Language (DDL) which provide CRUD operations at the level of relations themselves: creating (`CREATE TABLE`), updating (`ALTER TABLE`), and deleting relations(`DROP TABLE`) from the database.