Query languages provide support for retrieving information from a database.

Introduced the relational algebra:

- A procedural query language
- Six fundamental operations:
  - select, project, set-union, set-difference, Cartesian product, rename
- Several additional operations, built upon the fundamental operations:
  - set-intersection, natural join, division, assignment
Extended Operations

- Relational algebra operations have been extended in various ways
  - More generalized
  - More useful!
- Three major extensions:
  - Generalized projection
  - Aggregate functions
  - Additional join operations
- All of these appear in SQL standards
Would like to include computed results into relations
- e.g. “Retrieve all credit accounts, computing the current ‘available credit’ for each account.”
- Available credit = credit limit – current balance

Project operation is generalized to include computed results
- Can specify functions on attributes, as well as attributes themselves
- Can also assign names to computed values
- (Renaming attributes is also allowed, even though this is also provided by the ρ operator)
Generalized Projection

- Written as: $\Pi_{F_1, F_2, \ldots, F_n}(E)$
  - $F_i$ are arithmetic expressions
  - $E$ is an expression that produces a relation
  - Can also name values: $F_i$ as name

- Can use to provide derived attributes
  - Values are always computed from other attributes stored in database

- Also useful for updating values in database
  - (more on this later)
“Compute available credit for every credit account.”

\[
\Pi_{\text{cred}_i d, \ (\text{limit} - \text{balance}) \ \text{as available}\_\text{credit}} (\text{credit}\_\text{acct})
\]

<table>
<thead>
<tr>
<th>cred_id</th>
<th>limit</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-273</td>
<td>2500</td>
<td>150</td>
</tr>
<tr>
<td>C-291</td>
<td>750</td>
<td>600</td>
</tr>
<tr>
<td>C-304</td>
<td>15000</td>
<td>3500</td>
</tr>
<tr>
<td>C-313</td>
<td>300</td>
<td>25</td>
</tr>
</tbody>
</table>

\[credit\_acct\]
Aggregate Functions

- Very useful to apply a function to a collection of values to generate a single result
- Most common aggregate functions:
  - `sum`: sums the values in the collection
  - `avg`: computes average of values in the collection
  - `count`: counts number of elements in the collection
  - `min`: returns minimum value in the collection
  - `max`: returns maximum value in the collection
- Aggregate functions work on multisets, not sets
  - A value can appear in the input multiple times
"Find the total amount owed to the credit company."

\[ G_{\text{sum}}(\text{balance})(\text{credit\_acct}) \]

\[ 4275 \]

"Find the maximum available credit of any account."

\[ G_{\text{max}}(\text{available\_credit})(\prod (\text{limit} - \text{balance}) \text{ as available\_credit})(\text{credit\_acct}) \]

\[ 11500 \]
Grouping and Aggregation

- Sometimes need to compute aggregates on a per-item basis
- Back to the puzzle database:
  
  ```
  puzzle_list(puzzle_name)
  completed(person_name, puzzle_name)
  ```

- Examples:
  - How many puzzles has each person completed?
  - How many people have completed each puzzle?

<table>
<thead>
<tr>
<th>person_name</th>
<th>puzzle_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td>altekruse</td>
</tr>
<tr>
<td>Alex</td>
<td>soma cube</td>
</tr>
<tr>
<td>Bob</td>
<td>puzzle box</td>
</tr>
<tr>
<td>Carl</td>
<td>altekruse</td>
</tr>
<tr>
<td>Bob</td>
<td>soma cube</td>
</tr>
<tr>
<td>Carl</td>
<td>puzzle box</td>
</tr>
<tr>
<td>Alex</td>
<td>puzzle box</td>
</tr>
<tr>
<td>Carl</td>
<td>soma cube</td>
</tr>
</tbody>
</table>
Grouping and Aggregation (2)

“How many puzzles has each person completed?”

\[ \text{person}_\text{name} \bigcap \text{count}(\text{puzzle}_\text{name}) \text{(completed)} \]

- First, input relation \text{completed} is grouped by unique values of \text{person}_\text{name}
- Then, \text{count}(\text{puzzle}_\text{name}) is applied separately to each group
Grouping and Aggregation (3)

\[ \text{person}_{\text{name}} \ G_{\text{count}}(\text{puzzle}_{\text{name}}) (\text{completed}) \]

Input relation is grouped by \textit{person}_{\text{name}}

<table>
<thead>
<tr>
<th>person_{name}</th>
<th>puzzle_{name}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td>altekruse</td>
</tr>
<tr>
<td>Alex</td>
<td>soma cube</td>
</tr>
<tr>
<td>Alex</td>
<td>puzzle box</td>
</tr>
<tr>
<td>Bob</td>
<td>puzzle box</td>
</tr>
<tr>
<td>Bob</td>
<td>soma cube</td>
</tr>
<tr>
<td>Carl</td>
<td>altekruse</td>
</tr>
<tr>
<td>Carl</td>
<td>puzzle box</td>
</tr>
<tr>
<td>Carl</td>
<td>soma cube</td>
</tr>
</tbody>
</table>

Aggregate function is applied to each group

<table>
<thead>
<tr>
<th>person_{name}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
</tr>
<tr>
<td>Bob</td>
</tr>
<tr>
<td>Carl</td>
</tr>
</tbody>
</table>
Sometimes want to compute aggregates over sets of values, instead of multisets

Example:

- Change puzzle database to include a `completed_times` relation, which records multiple solutions of a puzzle

How many puzzles has each person completed?

- Using `completed_times` relation this time

<table>
<thead>
<tr>
<th>person_name</th>
<th>puzzle_name</th>
<th>seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td>altekruse</td>
<td>350</td>
</tr>
<tr>
<td>Alex</td>
<td>soma cube</td>
<td>45</td>
</tr>
<tr>
<td>Bob</td>
<td>puzzle box</td>
<td>240</td>
</tr>
<tr>
<td>Carl</td>
<td>altekruse</td>
<td>285</td>
</tr>
<tr>
<td>Bob</td>
<td>puzzle box</td>
<td>215</td>
</tr>
<tr>
<td>Alex</td>
<td>altekruse</td>
<td>290</td>
</tr>
</tbody>
</table>
“How many puzzles has each person completed?”

- Each puzzle appears multiple times now.

- Need to count distinct occurrences of each puzzle’s name

\[
person\_name \ G \ count\text{-distinct}(puzzle\_name)(completed\_times)
\]
Can append `-distinct` to any aggregate function to specify elimination of duplicates

- Usually used with `count`: \texttt{count-distinct}
- Makes no sense with `min`, `max`
General Form of Aggregates

- General form: \( G_1, G_2, ..., G_n \sum F_1(A_1), F_2(A_2), ..., F_m(A_m) (E) \)
  - \( E \) evaluates to a relation
  - Leading \( G_i \) are attributes of \( E \) to group on
  - Each \( F_j \) is aggregate function applied to attribute \( A_j \) of \( E \)

- First, input relation is divided into groups
  - If no attributes \( G_i \) specified, no grouping is performed
    (it's just one big group)

- Then, aggregate functions applied to each group
General Form of Aggregates (2)

- **General form:** $G_1, G_2, \ldots, G_n G_{F_1(A_1), F_2(A_2), \ldots, F_m(A_m)}(E)$

- **Tuples in $E$ are grouped such that:**
  - All tuples in a group have same values for attributes $G_1, G_2, \ldots, G_n$
  - Tuples in different groups have different values for $G_1, G_2, \ldots, G_n$

- Thus, the values $\{g_1, g_2, \ldots, g_n\}$ in each group uniquely identify the group
  - $\{G_1, G_2, \ldots, G_n\}$ are a superkey for the result relation
General form: \( G_1, G_2, ..., G_n \{ F_1(A_1), F_2(A_2), ..., F_m(A_m) \}(E) \)

- Tuples in result have the form:
  \( \{ g_1, g_2, ..., g_n, a_1, a_2, ..., a_m \} \)
  - \( g_i \) are values for that particular group
  - \( a_j \) is result of applying \( F_j \) to the multiset of values of \( A_j \) in that group

- **Important note:** \( F_j(A_j) \) attributes are unnamed!
  - Informally we refer to them as \( F_j(A_j) \) in results, but they have no name.
  - Specify a name, same as before: \( F_j(A_j) \) as attr_name
One More Aggregation Example

“How many people have completed each puzzle?”

\[ \text{puzzle}_\text{name} \ G_{\text{count}(\text{person}_\text{name})}(\text{completed}) \]

- What if nobody has tried a particular puzzle?
  - Won’t appear in \textit{completed} relation
One More Aggregation Example

- New puzzle added to `puzzle_list` relation
  - Would like to see `{ "clutch box", 0 }` in result...
  - "clutch box" won’t appear in result!

- Joining the two tables doesn’t help either
  - Natural join won’t produce any rows with "clutch box"

<table>
<thead>
<tr>
<th>puzzle_name</th>
<th>person_name</th>
<th>puzzle_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>altekruse</td>
<td>Alex</td>
<td>altekruse</td>
</tr>
<tr>
<td>soma cube</td>
<td>Alex</td>
<td>soma cube</td>
</tr>
<tr>
<td>puzzle box</td>
<td>Bob</td>
<td>puzzle box</td>
</tr>
<tr>
<td>clutch box</td>
<td>Carl</td>
<td>puzzle box</td>
</tr>
<tr>
<td></td>
<td>Alex</td>
<td>puzzle box</td>
</tr>
<tr>
<td></td>
<td>Carl</td>
<td>puzzle box</td>
</tr>
</tbody>
</table>

completed
Outer Joins

- Natural join requires that both left and right tables have a matching tuple
  \[ r \bowtie s = \prod_{R \cup S} (\sigma_{r.A_1=s.A_1 \land r.A_2=s.A_2 \land \ldots \land r.A_n=s.A_n}(r \times s)) \]

- **Outer join** is an extension of join operation
  - Designed to handle *missing information*

- Missing information is represented by *null* values in the result
  - *null* = unknown or unspecified value
Forms of Outer Join

- **Left outer join:** \( r \bowtie s \)
  - If a tuple \( t_r \in r \) doesn’t match any tuple in \( s \), result contains \( \{ t_r, \text{null}, \ldots, \text{null} \} \)
  - If a tuple \( t_s \in s \) doesn’t match any tuple in \( r \), it’s excluded

- **Right outer join:** \( r \bowtie s \)
  - If a tuple \( t_r \in r \) doesn’t match any tuple in \( s \), it’s excluded
  - If a tuple \( t_s \in s \) doesn’t match any tuple in \( r \), result contains \( \{ \text{null}, \ldots, \text{null}, t_s \} \)
Forms of Outer Join (2)

- **Full outer join:** $r \bowtie s$
  
  Includes tuples from $r$ that don’t match $s$, as well as tuples from $s$ that don’t match $r$

- **Summary:**

```
<table>
<thead>
<tr>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>r2</td>
<td>s2</td>
</tr>
<tr>
<td>c</td>
<td>r3</td>
<td>s3</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>r3</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>s2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>s4</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r1</td>
<td>null</td>
</tr>
<tr>
<td>b</td>
<td>r2</td>
<td>s2</td>
</tr>
<tr>
<td>c</td>
<td>r3</td>
<td>s3</td>
</tr>
<tr>
<td>d</td>
<td>null</td>
<td>s4</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r1</td>
<td>null</td>
</tr>
<tr>
<td>b</td>
<td>r2</td>
<td>s2</td>
</tr>
<tr>
<td>c</td>
<td>r3</td>
<td>s3</td>
</tr>
<tr>
<td>d</td>
<td>null</td>
<td>s4</td>
</tr>
</tbody>
</table>
```
Effects of *null* Values

- Introducing *null* values affects everything!
  - *null* means “unknown” or “nonexistent”
- Must specify effect on results when *null* is present
  - These choices are somewhat arbitrary…
  - (Read your database user’s manual! 😊)
- Arithmetic operations (+, –, *, /) involving *null* always evaluate to *null* (e.g. 5 + *null* = *null*)
- Comparison operations involving *null* evaluate to *unknown*
  - *unknown* is a third truth-value
  - **Note:** Yes, even *null* = *null* evaluates to *unknown*. 
Boolean Operators and unknown

- **and**
  
  \[ \text{true} \land \text{unknown} = \text{unknown} \]
  
  \[ \text{false} \land \text{unknown} = \text{false} \]
  
  \[ \text{unknown} \land \text{unknown} = \text{unknown} \]

- **or**
  
  \[ \text{true} \lor \text{unknown} = \text{true} \]
  
  \[ \text{false} \lor \text{unknown} = \text{unknown} \]
  
  \[ \text{unknown} \lor \text{unknown} = \text{unknown} \]

- **not**
  
  \[ \neg \text{unknown} = \text{unknown} \]
Relational Operations

- For each relational operation, need to specify behavior with respect to null and unknown

- **Select:** $\sigma_P(E)$
  - If P evaluates to unknown for a tuple, that tuple is excluded from result (i.e. definition of $\sigma$ doesn't change)

- **Natural join:** $r \bowtie s$
  - Includes a Cartesian product, then a select
  - If a common attribute has a null value, tuples are excluded from join result
  - Why?
    - $null = (anything)$ evaluates to unknown
Project and Set-Operations

- Project: $\prod(E)$
  - Project operation must eliminate duplicates
  - *null* value is treated like any other value
  - Duplicate tuples containing *null* values are also eliminated

- Union, Intersection, and Difference
  - *null* values are treated like any other value
  - Set union, intersection, difference computed as expected

- These choices are somewhat arbitrary
  - *null* means “value is unknown or missing”…
  - …but in these cases, two *null* values are considered equal.
  - Technically, two *null* values aren’t the same. (oh well)
Grouping and Aggregation

- In grouping phase:
  - *null* is treated like any other value
  - If two tuples have same values (including *null*) on the grouping attributes, they end up in same group

- In aggregation phase:
  - *null* values are removed from the input multiset before the aggregate function is applied!
    - Slightly different from arithmetic behavior; it keeps one *null* value from wiping out an aggregate computation.
  - If the aggregate function gets an empty multiset for input, the result is *null*…
    - …except for *count*! In that case, *count* returns 0.
Generalized Projection, Outer Joins

- **Generalized Projection operation:**
  - A combination of simple projection and arithmetic operations
  - Easy to figure out from previous rules

- **Outer joins:**
  - Behave just like natural join operation, except for padding missing values with *null*
Back to Our Puzzle!

“How many people have completed each puzzle?”

- Use an outer join to include all puzzles, not just solved ones
Counting the Solutions

- **Now, use grouping and aggregation**
  - **Group on puzzle name**
  - **Count up the people!**

\[
puzzle_{name} \sum_{person_{name}} \left(\text{puzzle list } \times \text{ completed}\right)
\]

<table>
<thead>
<tr>
<th>puzzle_name</th>
<th>person_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>altekruse</td>
<td>Alex</td>
</tr>
<tr>
<td>soma cube</td>
<td>Alex</td>
</tr>
<tr>
<td>puzzle box</td>
<td>Bob</td>
</tr>
<tr>
<td>altekruse</td>
<td>Carl</td>
</tr>
<tr>
<td>soma cube</td>
<td>Bob</td>
</tr>
<tr>
<td>puzzle box</td>
<td>Carl</td>
</tr>
<tr>
<td>puzzle box</td>
<td>Alex</td>
</tr>
<tr>
<td>soma cube</td>
<td>Carl</td>
</tr>
<tr>
<td>soma cube</td>
<td>null</td>
</tr>
<tr>
<td>clutch box</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>puzzle_name</th>
<th>person_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>altekruse</td>
<td>Alex</td>
</tr>
<tr>
<td>altekruse</td>
<td>Carl</td>
</tr>
<tr>
<td>soma cube</td>
<td>Alex</td>
</tr>
<tr>
<td>soma cube</td>
<td>Bob</td>
</tr>
<tr>
<td>soma cube</td>
<td>Carl</td>
</tr>
<tr>
<td>puzzle box</td>
<td>Bob</td>
</tr>
<tr>
<td>puzzle box</td>
<td>Carl</td>
</tr>
<tr>
<td>puzzle box</td>
<td>Alex</td>
</tr>
<tr>
<td>clutch box</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>puzzle_name</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>altekruse</td>
<td>2</td>
</tr>
<tr>
<td>soma cube</td>
<td>3</td>
</tr>
<tr>
<td>puzzle box</td>
<td>3</td>
</tr>
<tr>
<td>clutch box</td>
<td>0</td>
</tr>
</tbody>
</table>
Database Modification

- Often need to modify data in a database
- Can use assignment operator $\leftarrow$ for this

**Operations:**

- $r \leftarrow r \cup E$  
  Insert new tuples into a relation
- $r \leftarrow r - E$  
  Delete tuples from a relation
- $r \leftarrow \Pi(r)$  
  Update tuples already in the relation

- Remember: $r$ is a relation-variable
  - Assignment operator assigns a new relation-value to $r$
  - Hence, RHS expression may need to include existing version of $r$, to avoid losing unchanged tuples
Inserting New Tuples

- Inserting tuples simply involves a union:
  \[ r \leftarrow r \cup E \]
  - \( E \) has to have correct arity

- Can specify actual tuples to insert:
  \[ \text{completed} \leftarrow \text{completed} \cup \{ ("Bob", "altekruse"), ("Carl", "clutch box") \} \]
  - Adds two new tuples to \text{completed} relation

- Can specify constant relations as a set of values
  - Each tuple is enclosed with parentheses
  - Entire set of tuples enclosed with curly-braces
Inserting New Tuples (2)

- Can also insert tuples generated from an expression
- Example:
  
  “Dave is joining the puzzle club. He has done every puzzle that Bob has done.”

  - Find out puzzles that Bob has completed, then construct new tuples to add to `completed`
How to construct new tuples with name “Dave” and each of Bob’s puzzles?

- Could use a Cartesian product:
  \[
  \left\{ \text{“Dave”} \right\} \times \prod_{\text{puzzle\_name}} (\sigma_{\text{person\_name}=\text{“Bob”}}(\text{completed}))
  \]

- Or, use generalized projection with a constant:
  \[
  \Pi \text{“Dave” as person\_name, puzzle\_name}(\sigma_{\text{person\_name}=\text{“Bob”}}(\text{completed}))
  \]

Add new tuples to completed relation:

\[
\text{completed} \leftarrow \text{completed} \cup \Pi \text{“Dave” as person\_name, puzzle\_name}(\sigma_{\text{person\_name}=\text{“Bob”}}(\text{completed}))
\]
Deleting Tuples

- Deleting tuples uses the $\rightarrow -$ operation:
  \[ r \leftarrow r - E \]

- Example:
  Get rid of the “soma cube” puzzle.

Problem:
- completed relation references the puzzle_list relation
- To respect referential integrity constraints, should delete from completed first.
Deleting Tuples (2)

- completed references puzzle_list
  - puzzle_name is a key
  - completed shouldn’t have any values for puzzle_name that don’t appear in puzzle_list
  - Delete tuples from completed first.
  - Then delete tuples from puzzle_list.

\[
\text{completed} \leftarrow \text{completed} - \sigma_{\text{puzzle}_\text{name}=\text{“soma cube”}}(\text{completed}) \\
\text{puzzle}_\text{list} \leftarrow \text{puzzle}_\text{list} - \sigma_{\text{puzzle}_\text{name}=\text{“soma cube”}}(\text{puzzle}_\text{list})
\]

Of course, could also write:
\[
\text{completed} \leftarrow \sigma_{\text{puzzle}_\text{name} \neq \text{“soma cube”}}(\text{completed})
\]
In the relational model, we have to think about foreign key constraints ourselves...

Relational database systems take care of these things for us, automatically.

Will explore the various capabilities and options in a few weeks.
Updating Tuples

- General form uses generalized projection:
  \[ r \leftarrow \prod_{F_1, F_2, \ldots, F_n}(r) \]
  - Updates all tuples in \( r \)

- Example:
  - “Add 5% interest to all bank account balances.”
  \[ \text{account} \leftarrow \prod_{\text{acct_id, branch_name}, (\text{balance} \times 1.05)}(\text{account}) \]

  - Note: Must include unchanged attributes too
  - Otherwise you will change the schema of \( \text{account} \)
Updating Some Tuples

- Updating only some tuples is more verbose
  - Relation-variable is set to the *entire result* of the evaluation
  - Must include both updated tuples, and non-updated tuples, in result

- Example:

  “Add 5% interest to accounts with a balance less than $10,000.”

  \[
  \text{account} \leftarrow \Pi_{\text{acct_id, branch_name, (balance \times 1.05)}}(\sigma_{\text{balance} < 10000(\text{account})}) \cup \sigma_{\text{balance} \geq 10000(\text{account})}
  \]
Another example:

“Add 5% interest to accounts with a balance less than $10,000, and 6% interest to accounts with a balance of $10,000 or more.”

\[
\text{account} \leftarrow \Pi_{acct\_id,\ branch\_name,\ (\text{balance} \times 1.05)}(\sigma_{\text{balance} < 10000}(\text{account})) \cup \\
\Pi_{acct\_id,\ branch\_name,\ (\text{balance} \times 1.06)}(\sigma_{\text{balance} \geq 10000}(\text{account}))
\]

- Don’t forget to include any non-updated tuples in your update operations!
Relational Algebra Summary

- Very expressive query language for retrieving information from a relational database
  - Simple selection, projection
  - Computing correlations between relations using joins
  - Grouping and aggregation operations
- Can also specify changes to the contents of a relation-variable
  - Inserts, deletes, updates
- The relational algebra is a **procedural** query language
  - State a sequence of operations for computing a result
Benefit of relational algebra is that it can be formally specified and reasoned about.

Drawback is that it is very verbose!

Database systems usually provide much simpler query languages.

- Most popular by far is SQL, the Structured Query Language.

However, many databases use relational algebra-like operations internally!

- Great for representing execution plans, due to its procedural nature.
Next Time

- Transition from relational algebra to SQL
- Start working with “real” databases 😊