Partitioning and Aggregation

1 Preliminaries

1.a Relations, sets, and keys

You may recall from our overview of relational algebra that tuples in relations are unique, meaning that every tuple in a relation contains a unique combination of field values that distinguish it from all other tuples in the same relation.

In the general case, all attributes of a tuple are used to identify it, but often a subset of attributes suffices to uniquely identify every tuple in a relation. Such a subset of attributes is called a “key.” A key may contain a single attribute, all attributes, or any subset in between. If the key does not contain every attribute, we could treat the remaining non-key attributes as a “payload” of sorts: extra information associated with a tuple but that does not identify it (the key already does that). The division between key and non-key fields is similar to the key and payload pairs stored in a map or dictionary data structure. Note that relations can have multiple keys. For example, a license plate number (government identification) and VIN (manufacturer’s serial number) both uniquely identify a car.

Now consider the following two relations:

Students \((\text{student}_\text{id}, \text{surname}, \text{name})\)  
TAs \((\text{student}_\text{id}, \text{course}, \text{section})\)

The key of the Students table is highlighted in bold, while surname and name are “payload” (extra information about the student). Note that a student’s name is *not* a key, because two students could share the same name. The TAs table uses all attributes as its key; identifying the student uniquely is not enough, because they could have been a TA multiple times, even for the same course.

1.b Partitioning and partial keys

Now suppose we wanted to organize tuples from our TA relation into groups. The default grouping is by key: every tuple is its own entity, independent of other tuples in the relation. Sometimes it is useful to look at only some attributes when forming groups, though. What if we wanted to know how many courses a given student has tutored? Or how many tutors a given course has had? We can answer these questions by using “partial” keys. A partial key does not identify a tuple uniquely, and we do not want it to. Instead, a partial key identifies tuples that are “similar” or “related” because they share some of the same attributes.
Figure 1. Partitioning TA records (left) by student (middle) and by course (right)

Figure 1 shows a sample TA table partitioned in three different ways (with names given and section numbers omitted for clarity). The left column shows the default, which is to treat each tuple as its own entity, based on the full key. In the middle, we partition the table using the student name as a partial key, so that all tuples in the same group have the same student name. On the right, we partition using the course name as partial key instead. It turns out that partitioning is a powerful way to organize tuples for subsequent processing, as we will see soon.

<table>
<thead>
<tr>
<th>Student</th>
<th>Year</th>
<th>Dept</th>
<th>Course</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xiao</td>
<td>2009</td>
<td>CS</td>
<td>A08</td>
<td>B-</td>
</tr>
<tr>
<td>Xiao</td>
<td>2009</td>
<td>CS</td>
<td>A48</td>
<td>B</td>
</tr>
<tr>
<td>Xiao</td>
<td>2009</td>
<td>CS</td>
<td>A65</td>
<td>B+</td>
</tr>
<tr>
<td>Xiao</td>
<td>2009</td>
<td>Math</td>
<td>A23</td>
<td>B</td>
</tr>
<tr>
<td>Xiao</td>
<td>2009</td>
<td>Math</td>
<td>A30</td>
<td>B+</td>
</tr>
<tr>
<td>Xiao</td>
<td>2009</td>
<td>Math</td>
<td>A37</td>
<td>A</td>
</tr>
<tr>
<td>Xiao</td>
<td>2010</td>
<td>CS</td>
<td>B07</td>
<td>B</td>
</tr>
<tr>
<td>Xiao</td>
<td>2010</td>
<td>CS</td>
<td>B09</td>
<td>B-</td>
</tr>
<tr>
<td>Xiao</td>
<td>2010</td>
<td>CS</td>
<td>B36</td>
<td>B-</td>
</tr>
<tr>
<td>Xiao</td>
<td>2010</td>
<td>CS</td>
<td>B58</td>
<td>B</td>
</tr>
<tr>
<td>Xiao</td>
<td>2010</td>
<td>Math</td>
<td>B24</td>
<td>A-</td>
</tr>
<tr>
<td>Xiao</td>
<td>2010</td>
<td>Math</td>
<td>B41</td>
<td>B</td>
</tr>
<tr>
<td>Xiao</td>
<td>2010</td>
<td>Stats</td>
<td>B52</td>
<td>B-</td>
</tr>
<tr>
<td>Xiao</td>
<td>2011</td>
<td>CS</td>
<td>C24</td>
<td>B+</td>
</tr>
<tr>
<td>Xiao</td>
<td>2011</td>
<td>CS</td>
<td>C43</td>
<td>A-</td>
</tr>
<tr>
<td>Xiao</td>
<td>2011</td>
<td>CS</td>
<td>C69</td>
<td>A</td>
</tr>
</tbody>
</table>

Figure 2. Adding attributes to a partitioning key gives more, smaller partitions

In general, a relation can be partitioned by any attribute (or set of attributes); an empty partition key puts all tuples into the same giant partition, while a partition key equal to the schema puts every tuple in its own partition (recall that tuples in a relation are all unique). For example, Figure 2 shows three of the partitions that result when a partition key is empty (all courses), contains one attribute (year), or
contains two attribute (year, department). Note that, for clarity, the figure only shows one partition for each of the latter cases; there are actually three year partitions and six year-dept partitions.

1.c The MAP and REDUCE functions

Suppose we wish to compute a single value from a homogenous sequence of values (with all elements having the same type T). Examples might include computing a sum or average, but many computations have this property. A pair of higher-order functions¹ called map and reduce work together to perform this task. The map function performs a transformation on the sequence, modifying the elements and possibly changing their type. The reduce function then distills the elements into a single value.

The map function accepts two arguments as input: the sequence to transform and the “mapping function” \( g(T) \rightarrow R \) to be applied to each element. The output of map is another homogenous sequence, with values of type R; the input sequence is not altered, and R may be the same type as T (if the mapping function only changes the value). We might define map in python as follows:²

```python
def my_map(seq, g):
    rval = []
    for x in seq:
        rval.append(g(x))
    return rval
```

Assuming \( seq=[1, 2, 3, 4] \), example uses of map might include:

```python
my_map(seq, lambda x:x)
-> [1, 2, 3, 4] # identity, type unchanged
my_map(seq, lambda x:-x)
-> [-1, -2, -3, -4] # negate, type unchanged
my_map(seq, lambda x:.5*(x-2)*(x+1))
-> [-1.0, 0.0, 2.0, 5.0] # polynomial function, type is real
my_map(seq, lambda x:str(x))
-> ['1', '2', '3', '4'] # convert to strings
```

Mapping of an element is completely independent of any other element, so an implementation is free to re-order and parallelize the computation as it sees fit. Database engines, and map/reduce frameworks such as Hadoop, rely on this property for efficient processing of large datasets.

After map has prepared the sequence, reduce collapses it to a single value. The reduce function accepts three arguments:

- A homogenous sequence seq, whose elements all have type R (e.g. the output type of map).
- A function \( f(R,R) \rightarrow R \), used to combine two values into a single value.
- An optional value x (of type R), to be returned if the sequence is empty. It defaults to NULL.

¹ Higher order functions accept other functions as arguments
² Note that python provides a native map function with the same behavior. Also, python’s list comprehension syntax allows an even simpler definition: `my_map = lambda seq, g: [g(x) for x in seq]`
The reduce operation returns x if the sequence is empty. Otherwise, it remembers the first element of the sequence; all subsequent elements are passed to g() in turn, with the remembered result also passed in and replaced by the new return value. When processing reaches end-of-sequence, the remembered result is returned to the user.

For convenience, we will merge the functionality of map into reduce by allowing the latter to accept \( g(T) \rightarrow R \) as an optional fourth argument, and pass it an input sequence of type T instead of R.\(^3\) If we were to implement this augmented reduce in python,\(^4\) it might look like the following:

```python
def my_reduce(seq, f, g=lambda x:x, x=None):
    it = iter(seq)
    try:
        x = g(next(it))
        while 1:
            x = f(x, g(next(it)))
    except StopIteration:
        return x
```

Because it is a higher-order function, reduce is quite powerful and flexible; it’s probably easiest to show this by example. Suppose we have seq=[1,2,3,4]; and that we call my_reduce repeatedly with different functions for f and g:

```plaintext
my_reduce(seq, lambda x,y:x+y)
-> 10 # sum: ((1+2)+3)+4

my_reduce(seq, lambda x,y:x*y)
-> 24 # product: ((1*2)*3)*4

my_reduce(seq, lambda x,y:x+y, g=lambda x:x+1)
-> 4 # count: ((1+1)+1)+1

my_reduce(seq, lambda x,y:x+'-' y, g=lambda x:str(x))
-> '1-2-3-4' # string concatenation, similar to '-'.join(seq)

my_reduce(seq, lambda x,y:x*y, g=lambda x:x+1)
-> 120 # product of sums: (((1+1)*(2+1))*(3+1))*(4+1)
```

Thought experiment: how might you compute the average of a sequence using my_reduce?

Before we continue, it is worth noting that the function passed to reduce often has properties that allow an implementation considerable freedom in evaluating it. In the worst case, an arbitrary reducer function forces the implementation to evaluate all arguments one by one, serially (e.g. as shown in the above examples). An associative reducer, where \( f(x, f(y, z)) = f(f(x, y), z) \), allows some parallelism: neighboring pairs of elements can be reduced independently (cutting the list size in half), pairs of outputs can be reduced as well (again cutting the list size by half), and so on until only one value remains. All the reducing functions shown above are associative, and the effect of parallelism is easy to see with a longer input to string concatenation: [1,2,3,4,5,6,7,8] becomes ['1', '2', '3', '4', '5', '6', '7', '8'] (with the mapper), then ['1-2', '3-4', '5-6', '7-8'], then ['1-2-3-4', '5-6-7-8'], and finally ['1-2-3-4-5-6-7-8'].

---

\(^3\) Merging map into reduce is more concise, more efficient in practice (avoiding an intermediate result), and makes it easier to express the aggregate functions commonly used in SQL.

\(^4\) Note that python provides a built-in function called reduce that accepts the standard three arguments.
In addition to associativity, many reducer functions are *commutative*, where \( f(x, y) = f(y, x) \). When a reducer is both associative and commutative, reduce gives the same result no matter how the input is grouped or permuted for processing (as long as each element is processed exactly once). All aggregates specified in the SQL language are both commutative and associative, and database optimizers exploit this flexibility in several ways.

Finally, some reducers have the additional property of being *idempotent*, where the output of reduce depends only on what distinct values are in the input, not on how many times they occur. The \( \text{min} \) and \( \text{max} \) functions have this property, for example (you could even process the whole dataset twice, if you wanted, and still get the same result).

### 2 Introducing the grouping operator (\( \Gamma \))

With an understanding of partitioning and map/reduce, we are finally prepared to introduce the grouping operator used in relational algebra. The operator first partitions its input relation into “groups” according to a specified partitioning key (also known as the “grouping key”), and then applies a reduce operation within each group, using one reducer for each attribute other than the partial key used to create the partition. The output thus includes one tuple for every group, with each tuple’s grouping key attributes containing a combination of values that uniquely identifies the group that produced it; each of the remaining \( \text{aggregate} \) attributes contains the output of the reduce operation chosen for that attribute. Any attributes not mentioned are projected away.

Figure 3 illustrates the grouping operator in action, with grouping performed on column A and functions \( f_1 \) and \( f_2 \) specified as reducers for columns B and C. For each of the three partitions, the grouping key is copied to the output and the remaining fields are populated with the result of a reduce operation on the appropriate column of the partition.

---

5 Note that a function can be commutative without being associative. \( f(x, y) = (x+y)/2 \) is one such example: \( f(1, f(2, 3)) = 1.75 \) and \( f(f(1, 2), 3) = 2.25 \). Such functions do not allow meaningful reordering.
Note that grouping computes something about the partition as a whole rather than focusing on individual tuples. Aggregation combines multiple tuples within a partition into a single tuple that summarizes the group.

2.a Grouping operators in SQL

The SQL standard specifies several aggregate functions suitable for use with the grouping operator. These include min, max, count, sum, and average. Most database engines provide additional aggregates, as well as some mechanism for users to specify custom ones. Database aggregations are stateful, and are usually specified as three functions:

- **init()** initializes an internal state object and returns it to the database engine for safekeeping
- **step(state, x)** accumulates one additional value and returns the updated internal state
- **fini(state)** finalizes the internal state and returns the result of the aggregation

Together, these allow the same functionality as the map/reduce pair we discussed earlier, but the ability to maintain arbitrary internal state allows additional functionality. For example:

```python
# average
init = lambda: (0,0)
step = lambda s,x: (s[0]+x,s[1]+1)
fini = lambda s: s[0]/s[1] if s[1] else 0

# safe min
init = lambda: (False,None)
step = lambda s,x: (True,x if not s[0] or x < s[1] else s[1])
fini = lambda s: s[1]

# uniformly random element
init = lambda: (0,None)
def step(s,x):
    if (s[0] and random.randint(1,s[0]) > 1:
        return s[0]+1,s[1]
    return s[0]+1,x
fini = lambda s:s[1]
```

As an extreme case, advanced statistical aggregates might need to collect the entire partition, by calling step() on each input, before performing vector and matrix operations internally to arrive at a final result. Even more complex examples might consider multiple columns at a time (e.g. correlation testing). We will not consider such complex aggregates in this course, however.

2.b Aggregating aggregates

Recall that in relational algebra, the output of every operator is a relation that can serve as input to additional operators. This is true for grouping, which means (among other things) that we can perform grouping over previously-computed aggregations. As one example, consider the course listing in Figure 2: we can easily compute the average GPA for Xiao in each year (2009, 2010, 2011), using a grouping operator keyed by year and aggregating grade with the `average` function; given that result (containing three rows and two attributes) we might then compute the Xiao’s highest average yearly GPA (an empty grouping key, aggregating average_gpa with the `max` function); the corresponding relational algebra
expression would be: $\Gamma_{\text{max}(AG)}(\Gamma_{Y,AG=\text{avg}(G)}(R))$. This sort of nesting of aggregates could continue arbitrarily deep, but in practice there are seldom more than two aggregates in a row.

2.c Distributing work

Aggregation, and the map/reduce functions in general, provide a very convenient way to distribute work among many parallel processors for large-scale computation. Entire frameworks, most famously Hadoop, are built around this one concept. The reason map/reduce is so powerful comes from two reasons. First, partitioning creates many independent units of work that can be processed in parallel by different processors (or even computers). Second, map and reduce are extremely flexible, being higher-order functions. Aggregation, which turns a group into a single tuple, is only one possible type of reducer. A different reducer might extend every tuple in the group with some group-specific information, such as a running total or count within each group. Another reducer might run a multi-dimensional regression, converting a 2xN group into a 3x3 matrix describing that group’s behavior. The key is to format tuples appropriately and ensure that the partitioning key chosen groups together all tuples the computation might need (using a “map” phase) and then produce the actual answer desired (one “answer” per group) in a “reduce” phase. Compared with map/reduce frameworks, database engines provide more a complete infrastructure (other query operations and automatic optimization of the entire computation), but tend to restrict the kind of reducers that are allowed. The gap is closing, however. Map/Reduce frameworks increasingly provide query processing support (often in the form of SQL translators) while SQL was recently extended with so-called “window” functions that allow reducers other than aggregation, and provide more control over what calculations are applied to each group.