(1) Fetching data from disk.
(1a) Read 100 (page-sized) items sequentially:

We want to measure time cost. This is a linear-time algorithm, cost will
be completely dominated by disk I/O

```
// algorithm
seek the start of the sequence
for i := 1 to 100 do
{
  transfer the next item from disk
}

// cost estimation algorithm
cost := 0; // initialize accumulator, unit: miliseconds
cost += 10; // seek the start of the sequence
for i := 1 to 100 do
{
  cost += 0.1; //transfer the next item from disk
}
```

// equivalent closed-form representation
10 + 100*0.1 ms = 20 ms

(1b) Read 100 items in random order:

```
// algorithm
for i := 1 to 100 do
{
  seek the ith item
  transfer one item from disk
}

// cost estimation algorithm
cost := 0; // initialize accumulator, unit: miliseconds
for i := 1 to 100 do
{
  cost += 10; // seek the the ith item
  cost += 0.1; //transfer one item from disk
}
```

// closed-form solution
100*(10+0.1) ms = 1010 ms

(1c) Task: Enumerate all pairs from 100 items, stored on disk.
A RAM cache for 10 items is available.

Approach#1
upper bound, most naive random access implementation

```
//algo
for i in 1 to 100 do {
  seek the ith item;
  x := transfer;

  for j in 1 to 100 do {
    seek the jth item;
    y := transfer;

    output (x,y)
  }
}

// coarsening
cost := 0;
for i in 1 to 100 do {
  cost += 10+.1;
  for j in 1 to 100 do
    cost += 10+.1;
}
```

// closed-form solution
100*(1+100)*(10+0.1) ~10^5 ms

Approach#2 sequential access:

```
for i in 1 to 100 do {
  x := a[i] // seek pos i, transfer
  // seek pos 1
  for j in 1 to 100 do {
    // transfer
    y := a[j]
    output (x, y) // we assume this is an operation without
any I/O cost
  }
}

// coarsen into a function that computes I/O cost:
cost := 0;
for i in 1 to 100 do {
  cost += 10 // seek i
       +  .1 // transfer
       +  10 // seek 1
  for j in 1 to 100 do {
    cost += .1  // transfer
  }
}
```

// closed-form solution
100*(10+0.1+10+100*0.1) ~3000ms

Approach#3 sequential access + use 10-item sized cache:
We assume here that we can control the cache replacement policy (this is
not true for CPU caches, but for caches of secondary storage devices in RAM).

```
// fetch first 10 elements into cache, make sticky
// seek pos 1
for j0 in 1 to 10
  // transfer
  b[j0] := a[j0] // b is the cache
for i in 1 to 100
  x := a[i] // seek pos i, transfer
  for j0 in 1 in 10
     write (x, b[j0])
  // seek pos 1
  for j in 11 to 100
    // transfer
    y := a[j]
    write (x, y)

// coarsen into cost function:
cost := 0;
cost += 10;
for j0 in 1 to 10 do {
  cost += .1;
for i in 1 to 100 do {
  cost += 10 + .1 + 10;
  for j in 11 to 100 do
    cost += .1
  }
}

// closed-form
10 + 10*.1 + 100*(10+0.1+10 + 90*0.1) ~2900 ms
```

Approach #4 loop tiling, again we assume we control cache replacement
This will be block-nested loop join, which we will discuss in the second
lecture. The cost will be
10 * (2*10+ (10+100)*.1) = 310 ms!

(2) Task: selection sort of 100 items.
The input data array a[] is in RAM,
and we have a small cache, which is initially cold (empty).
The dominating cost here will be cache misses,
so that's what we are counting. This is acceptable because a cache miss
is equivalent to the cost of about 200 CPU instructions (70ns), and the
number of cache misses (at least in part a) is linear in the number of
operations.

```
(a)
for i in 1 to 100 do {
  for j in 1 to 100 do {
    // fetch a[i], cache miss only when j=1, assuming LRU
    // fetch a[j], cache miss nearly always
    if (a[j] < a[i]) swap a[i] with a[j]
  }
}

// closed-form
naively, ~100*(1+100)=10100 cache misses
```

(b) Loop tiling of selection sort suggests merge sort:

```
// 160 elements
// cache size is 10

//algorithm:
split up a[] into 16 runs r0[1] to r0[16] of 10 elements each
sort each runi r0[i] using selection sort.
// now we recursively merge sorted runs:
for i=1 to 8 do
  r1[i] := merge r0[i] and r0[8+i] // new run has 20 sorted elements.
for i=1 to 4 do
  r2[i] := merge r1[i] and r1[4+i] // new run has 40 sorted elements.
for i=1 to 2 do
  r3[i] := merge r2[i] and r2[2+i] // new run has 80 sorted elements.
r4[i] := merge r3[1] and r3[2]

// closed-form cost
16*10 + (8 + 4 + 2 + 1)*5*2
```

The assumption here is that we initially read the first five items of
each run into the cache, have a way of explicitly deallocating items from
the cache when they are not needed anymore, and assume that prefetching

does the rest in this very predictable scenario, so we have no further
cache misses in a merge run, and the cost per merge is 5*2.
Coming back to the assumption made above that we only need to count cache
misses, this is not really appropriate here: we assume the cost of a merge
constant (5*2), no matter how large the runs. (An experiment will probably
show that prefetching does not work perfectly and the cache misses of a
merge run grow linearly with the size of the runs, and still dominate
compute cost. If that's not true, model CPU cost more precisely!)

Note: We will look at sorting in more detail in week 2.

(3) Decide whether to introduce a new cache level into a CPU.
Cache size of the largest cache is currently c0.
Assume there are 8 cores, the new cache level would take the space of another
8 cores, and the workload is matrix multiplication. Does is make sense
to add the cache level or better to double the #cores?
That is, what is the best way of using this additional real estate on the chip?
Assume adding the cache level would multiply the size of the last level cache by 8, so the new last-level cache is of size 8*c0.

Assume two square matrices to multiply, each of size 8*c0

case 1: new cache level of size 8*c0; 8 cores

compute cost:
Loop tiling, put half of each matrix into cache.
We cut up the first matrix Aij along the i dimension and the second matrix Bjk along the k dimension.
let m be the cost of multiplying the matrix sequentially.
It is possible to do this such that after 2^2 steps we are done.
Multiplication of two half matrices: 1/2 * 1/2 * m
  -- this can be embarrassingly parallelized with the available 8 cores

sequential total time: 2^2 * (1/2)^2 * m / 8 s

case 2: double #cores to 16, cache size c0

compute cost:
loop tiling, put 1/16 of each matrix into cache.
It's possible to do this such that after 16*16 steps we are done.
Multiplication of two 1/16 matrices: 1/16 * 1/16 * m
  -- this can be embarrassingly parallelized with the available 16 cores

sequential total time: 16^2 * (1/16)^2 * m / 16 s

We are measuring only the CPU cost here, but not the cost of cache misses, so
the size of the caches does not matter (!), the assumption being that the
superlinear compute time dominates the cost of the cache misses. In the second
lecture we will look at this in more detail and take both the cost of compute
and slow memory into account.

=> 16 cores, not adding a cache wins
Setup:

- One computer, stores files on disk.
- Requests from other nodes are received and served via a network, Gbit Ethernet (i.e. 1/8 GB/s).
- Assume files are stored sequentially, 8KB pages, and already in compressed format.

Question 1. Describe in pseudocode the execution of a typical request in this architecture, and estimate the amount of time taken by each step. Assume the following times:

- marshalling/unmarshalling request data: 1ms
- sending a request throught the network: 10ms
- disk seek: 10ms
- disk read: 8KB/s
- network throughput: 1Gbit/s

How many queries per second can my computer serve?
Pseudocode of algorithm for serving one 128 MB file:
```
Receive request from client: // x2 (HTTP file
transfer requires two round trips)
```

```
  Client marshalls HTTP requests – headers,
contents (file path) // 1ms
  Send request through network // 10ms
  Unmarshal request // 1ms
  Server marshalls response, sends; client
demarshalls // 12ms
// ^ total time: 48ms
Fetch file content from disk:
  Seek file pages // 10ms
  Read file pages // 128MB/8KB*0.1ms = 1.6s
Send file through network // 128MB=(128/1000)GB
/ (1/8 GB/s) = 1sec
// Total time, sync: ~2.6s
```

Question 2. Make a simple sequence diagram showing the execution of the request. Can you suggest an improvement to this algorithm using multithreading?
That's too slow for me. I buy a new high-end disk with an order-of-magnitude faster throughput.

Diagram:

```
- Seek
| 10ms
|
- Read first page
| 0.1ms
|
- Send first page // (8KB=(8/1000/1000)GB) /
(1/8 GB/s)
| 0.064ms
|
- Read page n+1
| ...
// Total time, async: ~2.6sec
```

Better execution, using multithreading:

```
- Seek
| 10ms
|
- Read first page
| 0.1ms
|
- Read page n+1    - Send first page //
(8KB=(8/1000/1000)GB) / (1/8 GB/s)
| 0.1ms            | 0.064ms
|                  -
|
- Read page n+2   - Send page n+1
| ...             | ...
// Total time, async: ~1.6sec
```

Question 3. What could go wrong?
The service becomes popular but also slower, so it needs to scale!

Network becoming the bottleneck, geting saturated with packets. Useful question to ask: Is it really 1GB in guaranteed throughput between any two points at any time? Or is it the maximum throughput assuming no congestion at all?

Question 4. Suggest ways to scale this architecture up (open question).
Example approaches:

- Local redundancy (RAID-0 disk stripping)
- Distributed redundancy (slave servers)
    - Master forwarding files – dumb
    - request time: 1.6 + 1 + 1 = 3.6 sec
    - throughput: 1/3.6 /s
    - Master serving as directory is more useful
    - request latency *for server*: ~48ms
    - master server throughput: 1/0.048 = 21 /s
    - slave throughput: 1/1.6 = 0.63 /s
    - total throughput: min(21, N * 0.63)
    - master becomes bottleneck at N > 32

Considerations to have:

- What next?
- What if server fails?
- How to handle file changes?

If the files become too big: Partition the data, based on some hashing strategy to reduce ske; e.g., hash file identifier/path.

Question 5. Describe the difficulties that come with trying to replicating the master server.

The main difficulty is to have all replicates remain consistent, which can be very expensive if the data can changes, and if it may change often.

Exercise: Synchronous computations

In class, you have seen that synchronous communication/computation aligns *epochs* accross distributed systems.

Imagine a computation in which 5 servers perform some work in parallel, and synchronize at the end of an epoch before proceeding to the next one. At each epoch, the cost of server `i` at epoch `j` is `Cost(i, j)`.
Express the overall cost of the computation for N epochs as a sum of costs.
Answer: \sum_{i=1}^{N} max{Cost(j ,i) for all j in [1..5]}
Consider that the 5 servers still have to perform the same computations, with the same amount of data.
If the servers do not have to synchronize anymore, is it possible for the computation to finish earlier?
Answer: yes.
Could it be slower in certain cases?
Answer: no, except if you consider introducing overheads with extra communication among the servers.
Exercise: Distributed joins
Consider the following two entities:
Device
IoT id (8 bytes) | IoT type (50 bytes) | OS (4 byte)
Hacked
IoT id (8 bytes) | Severity (4 bytes) | Date (4 bytes)
The *Device* entity is on your local machine and contains a good 62 GB of data, while the *Hacked* is located on a distant machine and contains 1.2 TB of data.
Your goal is to check which of your machines has been hacked, the corresponding severity level of the hack, and see which OS was the most affected.
Semijoin filtering
In the case of a semijoin, what is the amount of data that you send to the remote machine?
Answer: 8 GB of data as we send the IoT id column.
In the case of a semijoin, and with 41 millions entries matching on the remote machine, what is the amount of data that you need to send back to your local machine?
Answer: 492 MB of data since we send back the severity along with the IoT id.
What if now the query runs on the remote machine and looks for the OS instead of the severity level? (Assume now that every one of your machines has been hacked once).
Answer: Send 600 GB from the remote to the local machine, then do the join, and send back 12 GB.
Does it make sense to execute the second query this way? What would be a more efficient implementation?
Answer: The two queries are equivalent in terms of logic, but the order and the machine on which we execute them matters as the second query is inefficient in this setup.
Bloom filter
You decide to optimize a little bit your system, and rely on bloom filters instead of semijoins.
Your friend Jeremy Mc. Hash hooks you up with a magic bloom filter that perfectly matches your needs.
Describe how the first query is executed now.
Answer: You send the bloom filter to the remote machine, compute the hacked entries that match the filter, and send them back.
How much data do you send in this case from the remote machine to your local machine? Give a lower bound.
Answer: the same as before or more due to false positives.
Is it possible for the bloom filter to send less data back to your machine than the semijoin? More?

Exercises 5

Data-parallel programming: exercises

Exercise 1.

Given `ls: List[Float]`, write a Scala expression equivalent to `ls.sum / ls.size` (i.e., computing the average of the values in the list) using only a single `foldLeft`, but no general recursion, and no other collection methods.

```
val sumCount = list.foldLeft((0.0, 0))((acc,
cur) => (acc._1 + curr, acc._2 +1))
```

```
sumCount._1 / sumCount._2
```

Exercise 2.

Given two binary relations R and S, each of type `List[(Int, Int)]`. Then the equijoin on `R._2 ===`

S._1 can be written using the Scala collections API (or, with minimal changes, Spark RDDs) as

```
R.flatMap(x => S.flatMap(y =>
  if (x._2 == y._1) List((x._1, x._2, y._2))
else List()))
```

We can call this the naive implementations of joins, akin to a naive tuple-by-tuple nested loops join.

Provide another implementation of joins, using Spark RDDs (but not using the RDD join operation) which differs from the above implementation in that it uses one of the key ideas of either block nested loops joins or GRACE hash joins. Using this idea should make your implementation typically faster on large datasets than the above naive implementation. Discuss and justify your solution.

Exercise 3.

Write a purely functional Scala program that computes the following query, respecting SQL's multiset semantics:

```
select * from R r1 where not exists
  (select * from R r2 where r1.A = r2.B)
```

(first try to understand what this does)

where binary relation R is represented in Scala by a value of type List[(Int, Int)] and A is represented by ._1 and B by ._2 of each tuple. The only permitted collection operations are list construction and flatMap. Do not define new recursive functions.

Homework: Now do it in Spark.

Exercise 4.

Given the following Scala data

```
val data: (List[Char],List[(String,Int)]) = (
  List('A','B'),
  List(("Aaron",42), ("Ron", 21), ("Bob",11),
("Arnold",11))
)
```

And consider the following query

```
data._2.filter(x =>
data._1.contains(x._1.head)).groupBy(_._2)
: Map[Int,List[(String, Int)]]
== Map(11 -> List((Bob,11), (Arnold,11)), 42 ->
List((Aaron,42)))
```

Give a relational algebra representation (with database relations and queries)

Homework: Give the a monad algebra representation.

Exercise 5.

Consider the following Scala data set:

```
val data: List[(String,List[Int])] =
List(("A",List(11,22)), ("B",List(2,3,4)))
```

And the following queries on it:

```
val q0: List[List[Int]] = data.map(_.2)
val q1 = q0.map(_.size)
val q2 = q1.max
```

Show how to represent each intermediate query in the relational model, along with the corresponding data tables.

Additional Homework

Consider the schema R(A, B), S(B, C, D), T(D, E) and the SQL query

```
select R.A, S.C, T.E from R, S, T where R.B =
S.B and S.D = T.D.
```

Write this query in monad algebra with equality. The input to this query is the input database given as a tuple of the three input relations, which each are a set of flat tuples, i.e. the input is of type

```
< R: {<A: Dom, B: Dom>}, S: {<B: Dom, C: Dom, D:
Dom>}, T: {<D: Dom, E: Dom>} >
```

Exercises 5 – Solutions

Data-parallel programming: exercise solutions

Exercise 1.

```
val sumCount = list.foldLeft((0.0, 0))((acc,
cur) => (acc._1 + curr, acc._2 +1))
sumCount._1 / sumCount._2
```

Exercise 2.

(Hashjoin-inspired Spark join, see Advanced Spark slides)

```
// the definition of two relations
val R: RDD[(Int, Int)]
val S: RDD[(Int, Int)]
// swapping the elements of individual tuples in
the records of R in order to put the key as the
first element
val Rm = R.map(r => r._2 -> r._1)
// specifies the number of partitions
val PARTS = 8
// partition both relations (the partitioning
phase of Grace-like hash join)
val Rp = Rm.partitionBy(new
org.apache.spark.HashPartitioner(PARTS))
val Sp = S.partitionBy(new
org.apache.spark.HashPartitioner(PARTS))
// zip the partitions and do a local join on
every node
val RSp = Rp.zipPartitions(Sp)((ri, si) =>
ri.flatMap(r => si.flatMap(s => if(r._1 == s._1)
List(r -> s) else Nil)))
RSp.collect
```

Exercise 3.

```
R.flatMap { r1 =>
  val matching =
  R.flatMap { r2 => if (r2._2 == r1._1) List(r2)
else List.empty }
  if (matching.isEmpty) List(r1) else List.empty
}
```

Exercise 4.

Letter	Value
	'A'
	'B'

Person	Name	Age
	"Aaron"	42
	"Ron"	21
	"Bob"	11

```
select p.Age, p.Name from Person p
  where substring(p.Name,1,1) in (select * from
Letter)
  group by p.Age
```

Exercise 5.

Here is the original (nested) data, viewed as a table:

data	_1	_2
	A	List(11,22)
	B	List(2,3,4)

Start by flattening it:

data_flat	_1	_2_flat
	A	11
	A	22
	B	2
	B	3
	B	4

The first query does nothing; it just conceptually hides one column.

```
q0 = select _1 as _1*, _2_flat from data_flat
```

q0	_1*	_2_flat
	A	11
	A	22
	B	2
	B	3
	B	4

```
q1 = select _1*, count(_2_flat) from q0 group by
_1*
```

q1	_1*	count(_2_flat)
	A	2
	B	3

```
q2 = select max(_1*) from q1
```

q2	max(_1*)
	3

Homework Help

Say we want to join two relations R and S, with the following record types:

```
R = < A: Dom, B: Dom >
S = < B: Dom, C: Dom, D: Dom >
```

That is, we're looking for a query `Q` with the following type:

```
Q: < r: {R}, s: {S} > -> {< A: Dom, B: Dom, C:
Dom, D: Dom >}
```

Query `Q` takes a record of two fields `r` and `s`, one for each collection of data. Remember that `{R}` means "a collection of items of type `R`".
So as expected, let us start by applying pair-with.

```
q0 = pairwith_r
q0: < r: {R}, s: {S} > -> {< r: R, s: {S} >}
```

The function `pairwith_r` "distributes" the elements of the collection associated with field `r`. For example, `pairwith_r(<r: {1,2,3}, x: "hello">) = {<r: 1, x: "hello">, <r: 2, x: "hello">, <r: 3, x: "hello">}`.
Then, we want to do the same pair-with on the elements of the other collection, so as to create a cartesian product, i.e., a combination of each element from `{R}` with each element from `{S}`. The problem is that now the `{S}` collection is *nested* inside another collection (notice the type returned by `q0` is wrapped in `{.}`). So in order to apply pair-with inside the element of the outer collection, we have to use `map`:

```
q1 = pairwith_r ∘ map(pairwith_s)
q1: < r: {R}, s: {S} > -> {{< r: R, s: S >}}
```

We now have a collection of collections of `R` and `S` pairs. We are not interested in the nesting structure, so we can flatten it:

```
q2 = pairwith_r ∘ map(pairwith_s) ∘ flatten
q2: < r: {R}, s: {S} > -> {< r: R, s: S >}
```

Now, this mapping + flattening is usually written with the `flatmap` shortcut, as below:

```
q2 = pairwith_r ∘ flatmap(pairwith_s)
```

The next step is to *project out* all the fields of the inner records, into a single top-level record. This is done by mapping a tuple of functions which projects out each field individually:

```
q3 = q2 ∘ map(< A: π_r ∘ π_A, B: π_r ∘ π_B,
Cr: π_r ∘ π_C, Cs: π_s ∘ π_C, D: π_s ∘ π_D
>)
q3: < r: {R}, s: {S} > -> {< A: Dom, B: Dom, Cr:
Dom, Cs: Dom, D: Dom >}
```

Remember that a tuple of functions applied to a value returns a tuple the result of applying each function to the value. For example `<f,g>(x) = <f(x),g(x)>`.
For brevity, let us define the type:

```
RS = < A: Dom, B: Dom, Cr: Dom, Cs: Dom, D: Dom
>
```

So we have:

```
q3: < r: {R}, s: {S} > -> {RS}
```

Now we want to filter out those records where the `Cr` component (the `C` value obtained from `r`) is the same as `Cs`. The first step is, for each element of the current collection, to create a record of the original record and of the condition `(Cr = Cs)`:

```
q4 = q3 ∘ map(< rs: id, eq: (Cr = Cs) >)
q4: < r: {R}, s: {S} > -> {< rs: RS, eq: {<>} >}
```

Notice the funny type of field `eq: {<>}` – it is either a collection of *the* empty tuple, or an empty collection. These two possible states encode the two boolean values.
Now we're almost done, as we've reduced the problem to filtering out records whose `eq` field is empty. This can be done easily using a pair-with followed by flattening:

```
q5 = q3 ∘ map(< rs: id, eq: (Cr = Cs) >) ∘
flatmap(pairwith_eq)
q5: < r: {R}, s: {S} > -> {< rs: RS, eq: <> >}
```

Indeed, records that have an empty `eq` (i.e., for which the condition is false), when paired-with `eq`, will return the empty collection: `pairwith_eq(< rs: <...>, eq: {} >) = {}`, while the other records will just contain a useless residual `eq` field:
`pairwith_eq(< rs: <...>, eq: {<>} >) = {< rs: <...>, eq: <> >}`,
Now all we need to do is to project the components we want to have the desired query shape:

```
Q = q5 ◦ map(< A: π_rs ◦ π_A, B: π_rs ◦ π_B, C: π_rs ◦ π_Cr, D: π_rs ◦ π_D >)
Q: < r: {R}, s: {S} > -> {< A: Dom, B: Dom, C: Dom, D: Dom >}
```

And we're done.
It should be rather straightforward to go from here to the full homework solutions.
Last modified: Sunday, 31 March 2019, 18:53