\[- \left( 1 - \left( 1 - \frac{1}{N} \right)^{\frac{RM}{N}} \right)^k \approx \left( 1 - e^{-\frac{RM}{N}} \right)^k \]

\[p(\text{any given bit not set by 1 hash})\]

\[p(\text{any given bit will remain 0 after } k \text{ hash calls for 1 record})\]
\[ R = \# \text{ of hash fans} \]
\[ M = \# \text{ of records} \]
\[ N = \# \text{ of buckets} \]
\[ c = \text{const} \]

\[ R = c \frac{M}{N} \]

\[ N \frac{M}{N} = c \frac{M}{R} \]
\[ y = (1 - e^{-\frac{1}{3}x})^x \]

\[ y = (1 - e^{-\frac{1}{10}x})^x \]

\[ y = (1 - e^{-\frac{1}{20}x})^x \]

\[ y = (1 - e^{-\frac{1}{30}x})^x \]

\[ y = \left(1 - e^{-\frac{1}{10^x}}\right)^x \]

\[ \frac{M}{N} = 5 \]

\[ \frac{N}{S} = 10 \]

\[ \sum = 20 \]

\[ p \]

Of a false positive

\[ R = C \frac{M}{N} \]

\[ n = 24 \]

\[ \text{Hof records track files} \]
Join Results
Matching Tuples

\[ R \]

\[ \sum_{i=1}^{n} \]

Scan
Compute Nodes

Units of computation

Input data

Input

Output

Optimization properties
- Latency/Time
- Throughput
- Data transfer
Parallelism Concepts

Terms:

- # of Cores
- Resources available to each core
  - Resources Shared between each core

Communication Model

- Shared Memory
  - Everyone can read from/write to the same address space
- Non-Uniform Memory Access
  - As shared memory, but explicit that some regions of memory (known in advance) can be accessed faster.
- Shared Disk
  - Each core has its own local resources (e.g., RAM), and a shared resource. (similar to NUMA)
- Message-Passing
  - aka “Shared Nothing”
  - Each core has its own local resources, and must explicitly send messages to other nodes
  - All models are equivalent in terms of expressive power, but differ in how “aware” the user needs to be about the cost of coordination when designing a system. Shared memory = 0% awareness, Message passing = 100% awareness

Memory Hierarchy

- Network, HDD, SSD, RAM, L1 Cache, L2 Cache, L3 Cache

Parallelism Models

- Multi-Core CPUs
  - (typically) Shared L2 cache
  - On-Chip Interconnect
- Multi-CPU Devices
  - Shared RAM
  - Motherboard Interconnect
- Multi-Node
  - Network interconnect only

Operator Parallelism

How do we subdivide a task (AB)

- Option 1: Data Parallelism
  - AB1: Run AB on half the data
  - AB2: Run AB on the other half of the data

- Option 2: Pipeline Parallelism
- Step A produces outputs 1 at a time
- Step B consumes A's outputs

**Communication**

**Data Parallelism**
- AB1 and AB2 don’t communicate (assumed to have all data upfront)

**Pipeline Parallelism**
- A sends everything to B

**Both**
- A * (B1 + B2)
  - Possibility 1: A sends everything to both B1, B2
  - Possibility 2: A sends some things to B1, some to B2
- (A1 + A2) * B
  - Only Possibility: A1, A2 both send everything to B (Fold/Reduce)
- (A1 + A2) * (B1 + B2)
  - Possibility 1: A1 sends everything to B1, A2 to B2 (Map)
  - Possibility 2: A1,A2 send some things to B1, some to B2 (Shuffle)
  - Possibility 3: A1,A2 send everything to both B1,B2

**Storm Model**
- Two types of Operators
  - Spout = Data Source
  - Bolt = Operator
- Workflow definition declares...
  - A parallelism level for each bolt
  - A set of pipes linking bolts
- Bolts see a set of input and output pipes
  - Bolts not called explicitly: just read from their pipes.
  - Bolts manually determine which pipe to send data into

**Map/Reduce Model**
- Map task (purely parallel)
  - Code that reads 1 record (at a time), and produces any number of key/value pairs
- Shuffle (internal process)
  - k/v pairs grouped by keys
- Reduce task
  - Code that reads 1 key + an iterator over values with that key
- Combine Task
  - A “pre-reduce” step where values for the same key are “combined” (see Aggregates, below)
  - E.g., word count example?
Partitioning

What is one “fragment” of data?
- Logical unit of data/computation
  - E.g., A Tuple.

How do we decide which logical unit(s) of data are grouped together (buckets)?
- Partitioning Strategy 1: Random
- Partitioning Strategy 2: By Range
  - Hard to balance the size of each bucket
- Partitioning Strategy 2: By Hash
  - Effectively random for range lookups
  - Remains unbalanced if some records are “common”
  - Similar issues as indexing

IO is Sloooooooow
- Each Message/Write is an overhead
- Goal: Minimize data transferred

RA Operators

Select, Project, Union
- Logical Unit of Data: 1 tuple
- No data dependencies between tuples

Aggregate
- Logical Unit of Data: 1 group
  - Reduce Messy! No parallelism

But can do better with algebraic aggregates
- Fan-in aggregation
  - E.g. \( \text{SUM}(A, B, C, D, \ldots) = (A + B) + (C + D) + \ldots \)
  - Compute \( x = A + B, y = C + D, z = \ldots \)
  - Compute \( x + y + z \)
  - Makes a “fan-in” tree. Log compute required vs Lin compute

Join
- Logical Unit of Data: 1 tuple^2
- No data dependencies between tuple pairs
- … but can potentially rule out some candidate tuple pairs

How much data needs to be transferred?
- \( R[1\ldots N] \times S[1\ldots M] \) partitions: \( R[1] \) cloned \( M \) times, \( S[1] \) cloned \( N \) times (Total Data: \( N \times M + M \times N \))
- We can do better…
Data Partitioning
- Hash Grid for EQ joins
- Range Grid for InEQ joins

Bloom Join

Central Idea: Eq Joins are very selective
- A LHS row with a join key that has no match on the RHS is wasted data transfer

Tactic 1: Have the RHS send the LHS a list of its keys
- Big! Potentially lots of data being transferred
  - 1 int = 4/8 bytes of data
  - LINEITEM @ SF 1 = 6m Ints = 24/48MB
  - Can we do something smaller?

Tactic 2: Parity bit
- Split keys into 2 groups (e.g., by a hash)
  - RHS says whether there are any matching keys in group 1, and whether any in group 2
  - 2 bits total!
  - Good… but useless after both bits set

Tactic 3: Parity bits
- Split keys into N groups
  - Better, requires N bits!
  - Good… but becomes useless quickly
    - Every new tuple on the RHS has a 1/N chance to trigger a false positive for each row of the LHS
    - Can we reduce the chance of a false positive further?

Tactic 4: Bloom filters
- Assign each key into k / N groups
  - Still only requires N bits
  - Use k hash functions to pick which groups a key goes into (groups sampled with replacement ok)
  - Oddly enough, becomes useless far more slowly
    - Can rule out membership if ANY of the k/N group bits aren’t set.
    - Need k/N tuples in RHS to align to trigger a false positive (much lower chance, see below).

Some Math:
- Probability that 1 bit is set by 1 hash fn: 1/N
- Probability that 1 bit is not set by 1 hash fn: 1-1/N
- Probability that 1 bit is not set by k hash fns: (1-1/N)^k
- … for m separate records: (1-1/N)^km
- Probability that 1 bit is set by k hash fns for m records: 1 - (1-1/N)^km
Probability that all k bits are set: $(1 - (1-1/N)^{km})^k$

- or approximately $(1-e^{(-km/N)})^k$

- The probability of a false positive, aka collision

- Minimal $P[collision]$ is at $k \approx c \cdot \frac{m}{n}$