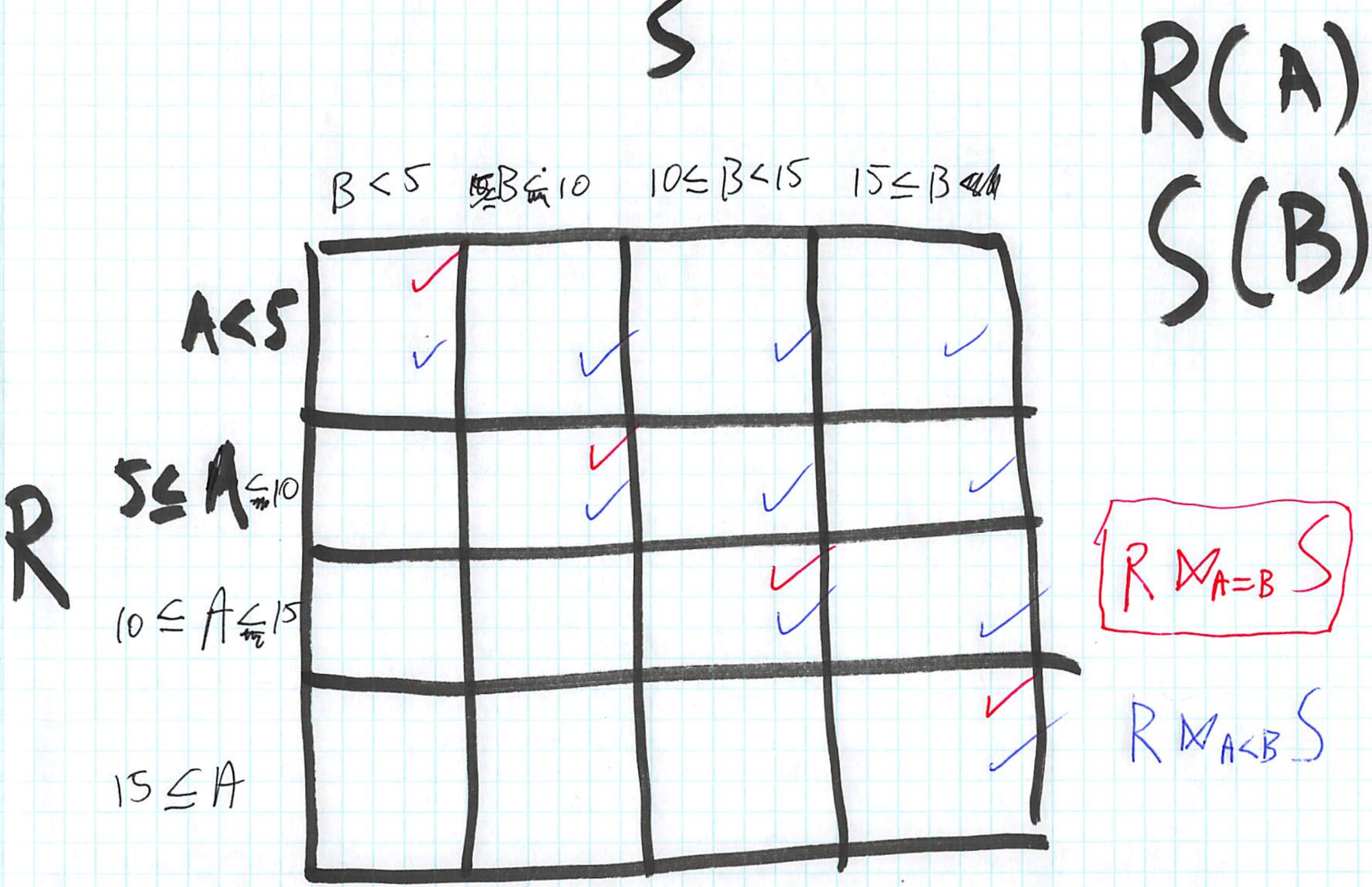
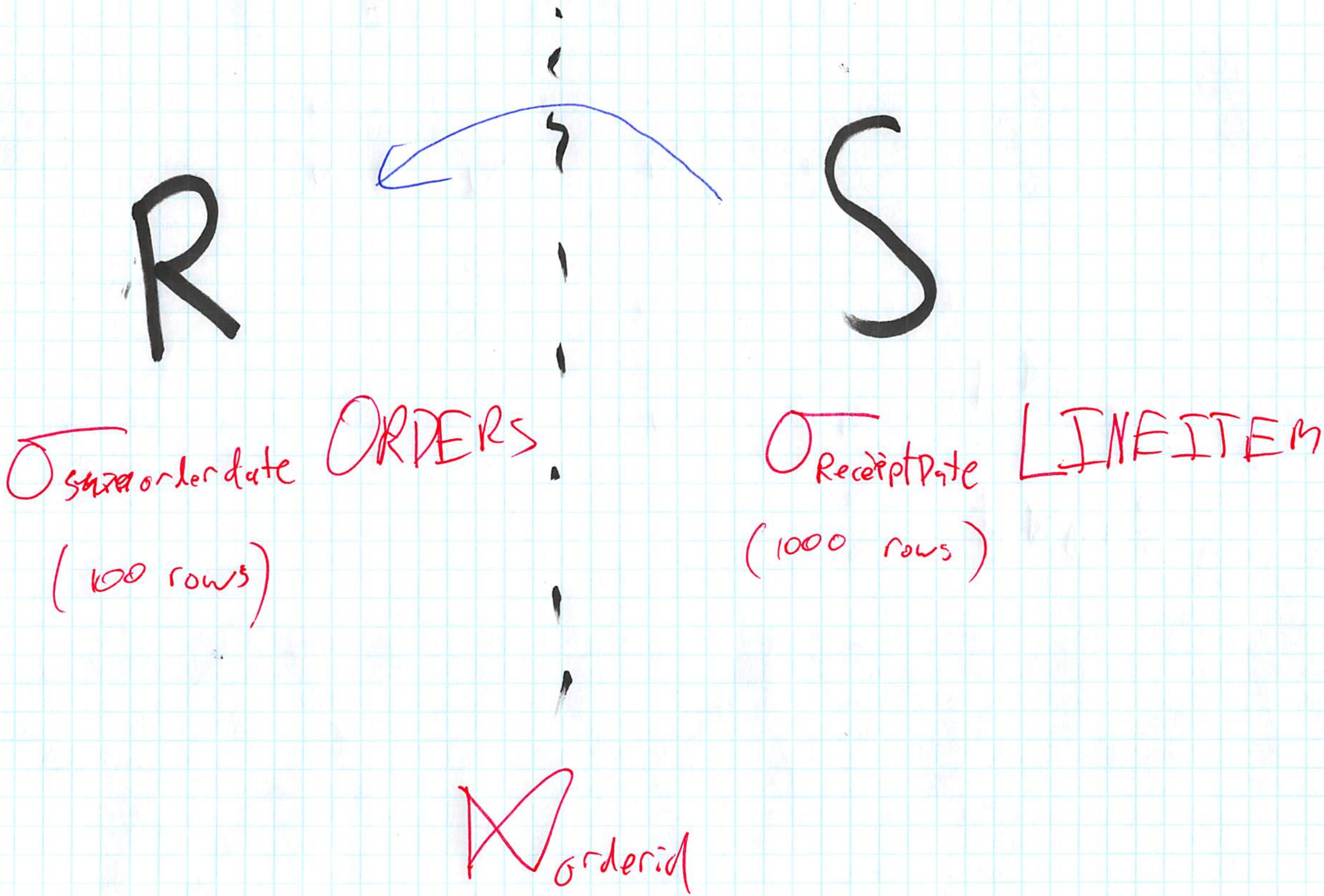
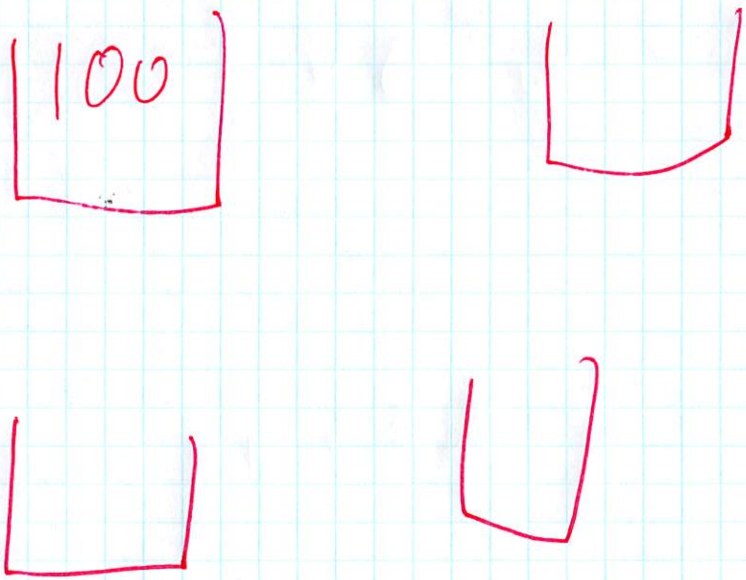

S
R(A)
S(B)
B < 5
B ≤ 10
10 ≤ B < 15
15 ≤ B
R
A<5
5 ≤ A ≤ 10
10 ≤ A ≤ 15
15 ≤ A
R ⋈A=B S
R ⋈A<B S


R
S
σ orderdate ORDERS
(100 rows)
σ ReceiptDate LINEITEM
(1000 rows)
⋈ orderid



R	A		
	1 →	1	1
	3 →	1	3
	5 →	1	1
	7 →	1	3
	11 →	1	3
	2 →	0	2

$h(x) = x \% 4$

S	B
	2 → 0
	3 → 1
	4 → 0
	5 → 1
	6 → 0

p(1 bit set)

$$1-\left(1-\left(1-\frac{1}{N}\right)^{km}\right)^{k} \approx \left(1-e^{-\frac{km}{N}}\right)^{k}$$

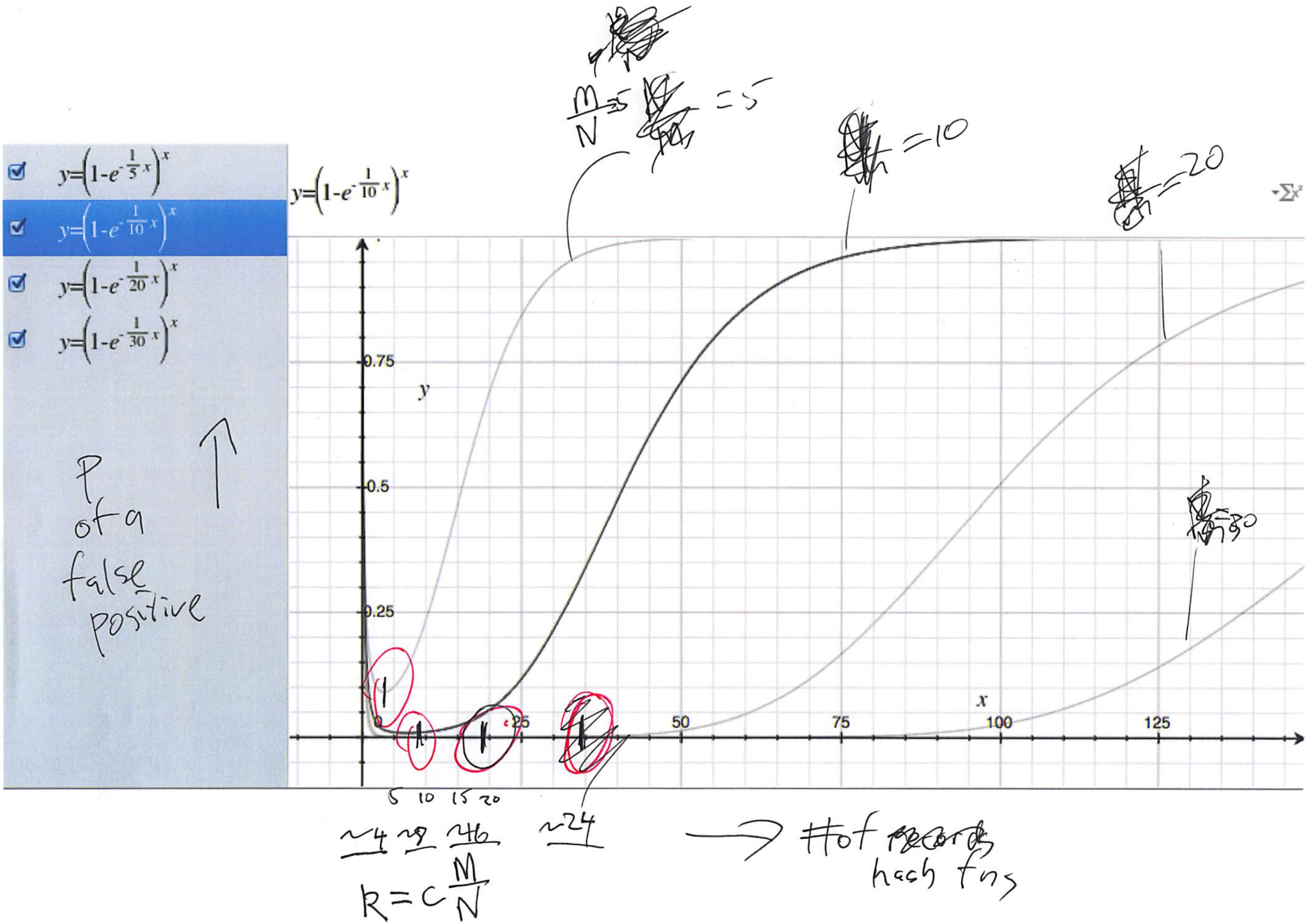
p(any given bit not set by 1 hash)

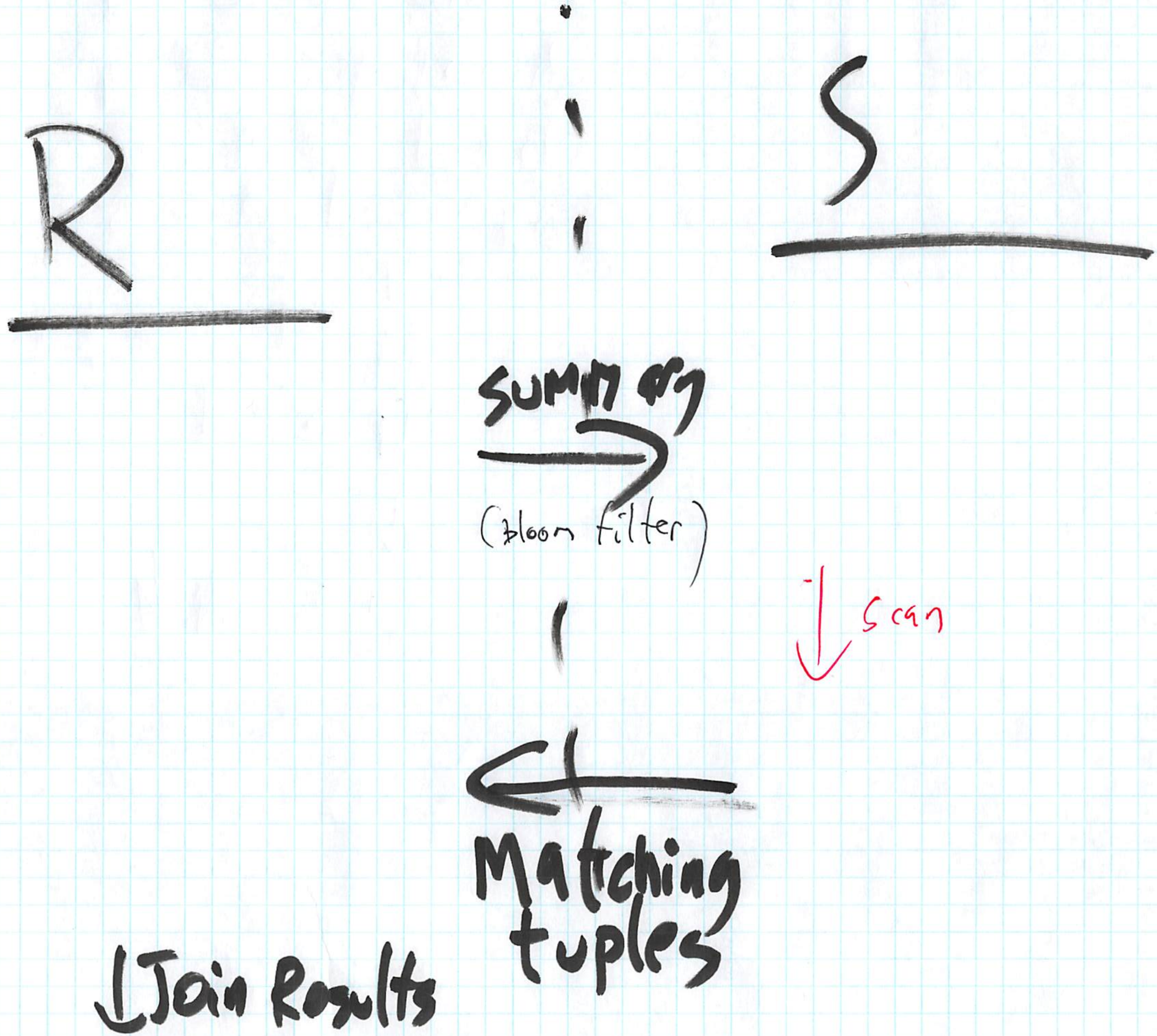
p(any given bit will remain 0 after k hash calls for 1 record)

k = # of hash fns
M = # of records
N = # of buckets
c = const

$$k \approx c \frac{M}{N}$$

$$N \approx c \frac{M}{k}$$

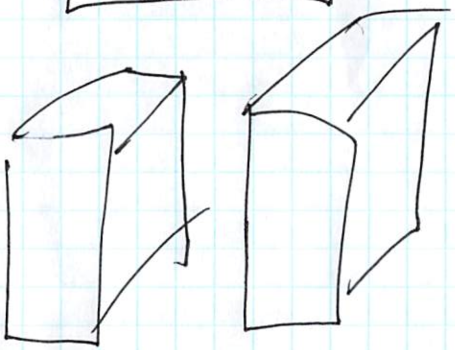

y=(1-e^(-1/5 x))^x
y=(1-e^(-1/10 x))^x
y=(1-e^(-1/20 x))^x
y=(1-e^(-1/30 x))^x
y=(1-e^(-1/10 x))^x
0.75
0.5
0.25
y
x
25
50
75
100
125
P of a false positive
M/N
=5
=10
=20
5 10 15 20
~4
~24
R = c M/N
# of hash fns


R
S
summary
(bloom filter)
scan
Matching tuples
Join Results

Compute Nodes

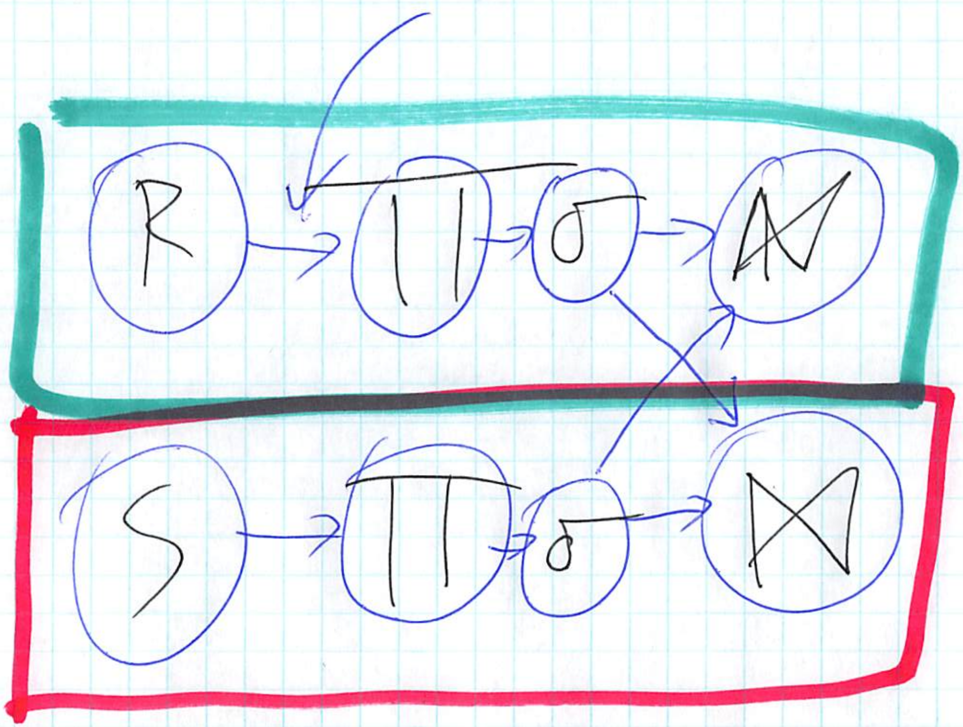
Units of computation

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 ofOperators
    - 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. SUM(A, B, C, D, …) = (A + B) + (C + D) + …
      - Compute x = A+B, y = C+D, z = ...
      - 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…N] x S[1…M] partitions: R[1] cloned M times, S[1] cloned N times (Total Data: NxM + MxN)
    - 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 bit***s***

- **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 $k \approx c \cdot m/n$