A Generalized Birthday Problem

David Wagner

University of California at Berkeley

The classic birthday problem

Given two lists L_1, L_2 of elements drawn uniformly and independently at random from $\{0,1\}^n$, find $x_1 \in L_1$ and $x_2 \in L_2$ such that $x_1 \oplus x_2 = 0$.

Hashing

- Hash function $h: \{0,1\}^* \rightarrow \{0,1\}^n$
- Lists L_1, L_2
- j-th element of L_i is $h(i,j)$
- For each $x_1 \in L_1$ and $x_2 \in L_2$, look for $x_1 = x_2$ (same as $x_1 \oplus x_2 = 0$)
- A solution immediately yields a collision
- Generate elements of lists until you find a collision

The join operation

DEFINE: S ⋈ T returns a list of elements common to both S and T

Solving the classic birthday problem

- $x_1 \in L_1$, $x_2 \in L_2$
- Since $x_1 \oplus x_2$ iff $x_1 = x_2$, we can solve by simply computing $L_1 \bowtie L_2$

Solving the classic birthday problem

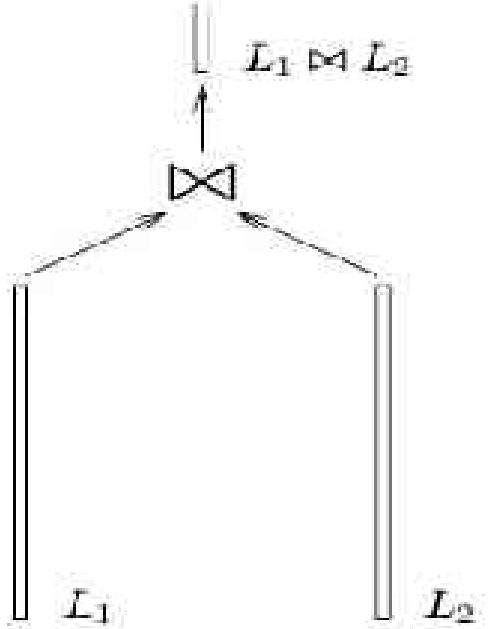


Fig. 1. An abstract representation of the standard algorithm for the (2-list) birthday problem: given two lists L_1, L_2, we use a join operation to find all pairs (x_1, x_2) such that $x_1 = x_2$ and $x_1 \in L_1$ and $x_2 \in L_2$. The thin vertical boxes represent lists, the arrows represent flow of values, and the $\bowtie$ symbol represents a join operator.

Solving the classic birthday problem

- Efficient methods for computing joins (database query evaluation)
- n-bit values: $O(2^{n/2})$ time/space
- Assumes we are free to choose the size of the lists however we like

The k-sum problem

Given k lists L_1, ..., L_k of elements drawn uniformly and independently at random from $\{0,1\}^n$, find $x_1 \in L_1$, ... , $x_k \in L_k$ such that $x_1 \oplus x_2 \oplus \ldots \oplus x_k = 0$.

Specifically, the 4-sum problem

- $L_1, ..., L_4$ lists
- $x_i \in L_i$ such that $x_1 \oplus x_2 \oplus x_3 \oplus x_4 = 0$
- Solution should exists with good probability if each list has length at least $2^{n/4}$
- But...the most obvious approaches require $2^{n/2}$ steps of computation
- This paper develops a more efficient algorithm, first for k=4, then for arbitrary k

Definitions

- $\text{low}_l(x)$ = the l least significant bits of x
- $L_1 \bowtie_l L_2$ = all pairs from $L_1 \times L_2$ that agree in the l least significant bits

Observations

- $low_l(x_i \oplus x_j) = 0$ iff $low_l(x_i) = low_l(x_j)$
- Given L_i, L_j, we can easily generate all pairs $\langle x_i, x_j \rangle$ satisfying $x_i \in L_i$, $x_j \in L_j$, and $low_l(x_i \oplus x_j) = 0$ by using the $\bowtie_l$ operator
- If $x_1 \oplus x_2 = x_3 \oplus x_4$, then $x_1 \oplus x_2 \oplus x_3 \oplus x_4 = 0$

Observations

- If $\text{low}_l(x1 \oplus x2) = 0$ and $\text{low}_l(x3 \oplus x4) = 0$, then we necessarily have $\text{low}_l(x1 \oplus x2 \oplus x3 \oplus x4) = 0$, and also $\Pr[x1 \oplus x2 \oplus x3 \oplus x4 = 0] = 2^l / 2^n$

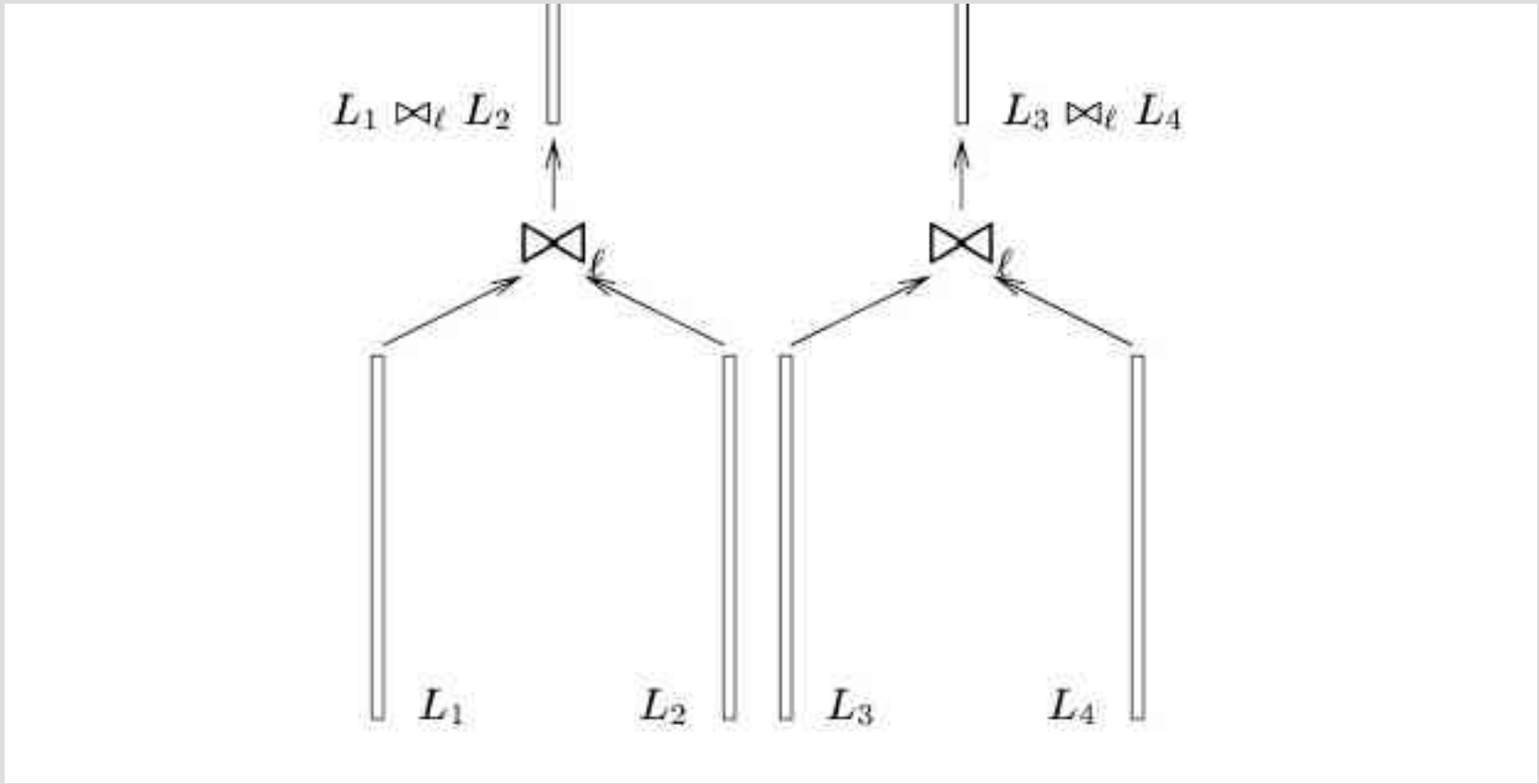
Solving 4-sum (step 1)

- Extend L_1, ..., L_4 by adding candidate values until each list contains about 2^l elements
- (l chosen in a specific way, described later)

Solving 4-sum (step 2)

- Use the $\bowtie_l$ operator
- Generate L_{12} of values $(x_1 \oplus x_2)$ such that $low_l(x_1 \oplus x_2) = 0$
- Generate L_{34} of values $(x_3 \oplus x_4)$ such that $low_l(x_3 \oplus x_4) = 0$

Solving 4-sum (step 2)



Solving 4-sum (step 3)

- So L_{12} is made up of $(x_1 \oplus x_2)$ elements
- L_{34} is made up of $(x_3 \oplus x_4)$ elements
- Run $L_{12} \bowtie L_{34}$ (comparing all bits)
- Previous observation: If $(x_1 \oplus x_2) = (x_3 \oplus x_4)$, then $x_1 \oplus x_2 \oplus x_3 \oplus x_4 = 0$
- Therefore, $L_{12} \bowtie L_{34}$ yields a solution to the problem

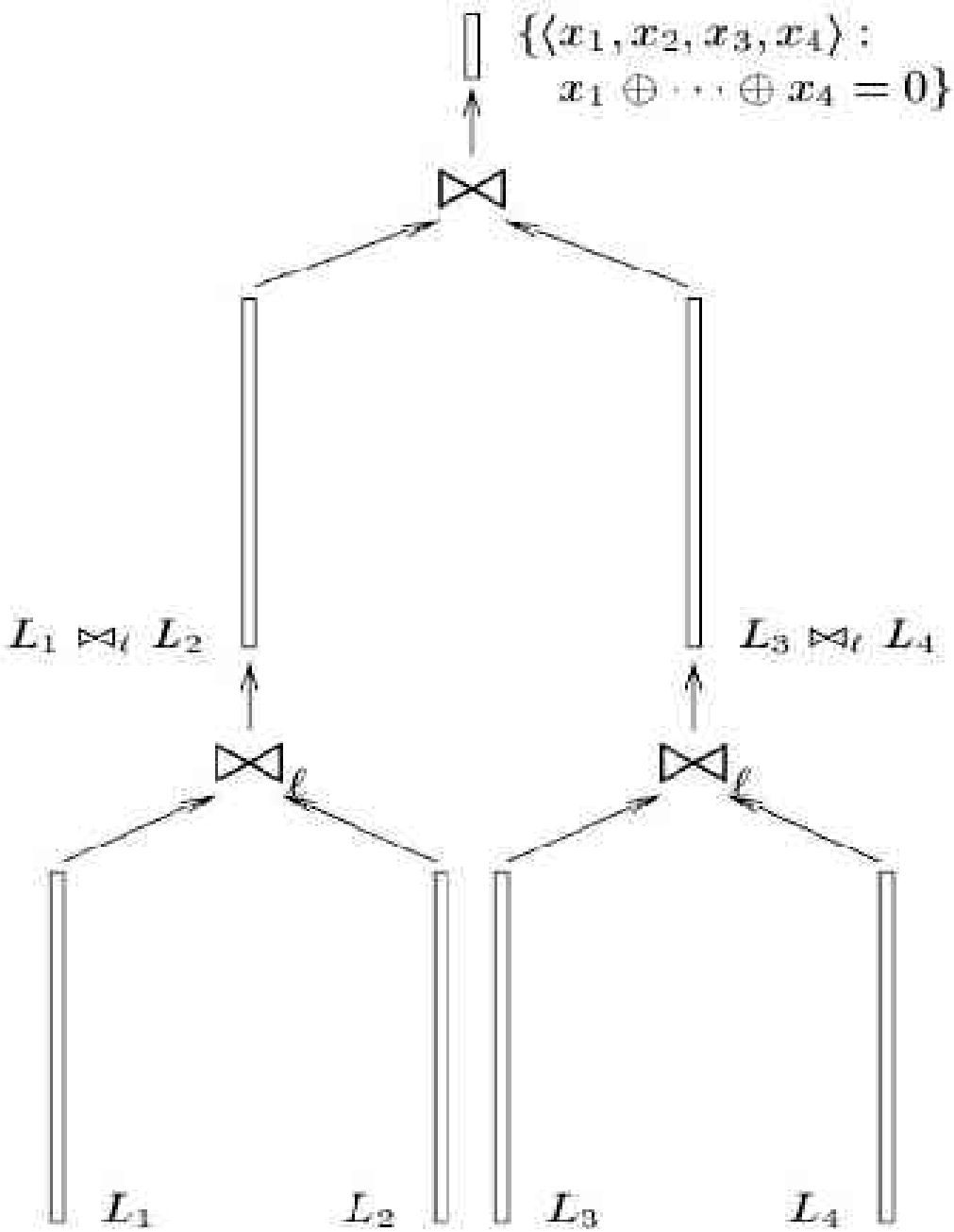


Fig. 2. A pictorial representation of our algorithm for the 4-sum problem.

Complexity (or “how we choose l”)

- x_1, x_2 chosen uniformly at random
  - $\Pr[\text{low}_l(x_1 \oplus x_2) = 0] = 1/2^l$
- Expected value of $|L_{12}|$ is

  $|L_1| \times |L_2| / 2^l = 2^{2l} / 2^l = 2^l$
  - Same for $|L_{34}|$
- Any one pair of elements from $L_{12} \times L_{34}$ matches with probability $2^l / 2^n$ (previous observation)

Complexity

- Expected number of elements in common is $|L_{12}| \times |L_{34}| / 2^{n-l} = 2^{n-3l}$
- This is at least 1 when $l \geq n/3$
- So set $l = n/3$
  - So the size of each list is $2^{n/3}$
- Therefore the algorithm is $O(2^{n/3})$
  - Recall previous best was $O(2^{n/2})$

An aside

- Special property of the algorithm
  - Only solutions where $(x_1 \oplus x_2)$ and $(x_3 \oplus x_4)$ are zero in the low l bits are found
- This isn't a requirement
- To sample from the set of all solutions
  - Pick a random l-bit value α
  - Compute $(L1 \bowtie_l (L2 \oplus \alpha)) \bowtie (L3 \bowtie_l (L4 \oplus \alpha))$

Another aside

- In $x_1 \oplus x_2 \oplus x_3 \oplus x_4 = 0$, the value 0 is not special
- $x_1 \oplus x_2 \oplus x_3 \oplus x_4 = c$, c a constant, is valid without any extra complexity

Our complexity can't get worse

- For k > k', we can reduce a k-sum problem down to a k'-sum problem
- Pick arbitrary values for $x_{k'+1}$, ..., x_k from $L_{k'+1}$, ..., L_k
- Set $c = x_{k'+1} \oplus ... \oplus x_k$
- Use k'-sum to solve $x_1 \oplus ... \oplus x_{k'} = c$
- Paper calls this the “list-elimination trick”
- Complexity can only get better or stay the same as k increases

Our complexity can get better

- We can do better than cube-root time for larger values of k
- When k is a power of 2, we use a complete binary tree of depth lg k
- At height h of the tree, we let
  $l_h = hn / (1 + \lg k)$, and use $\bowtie_{l\_h}$ to filter
- At root we use the full join
- At each inner node value x, $x = x' \oplus x''$, where x' and x'' are from child lists

Upper bounds for higher k

- k-sum uses lists of size $O(2^{n/(1+\lg k)})$
- Uses $O(k \bullet 2^{n/(1+\lg k)})$ time and space
- Slow increase until k becomes large

When k is not a power of 2

- Take k' to be the largest power of 2 less than k
- Run the list elimination trick to go from a k-sum problem to a k'-sum problem
- This doesn't run any faster for $k = 2^i + j$ than it does for $k = 2^i$
- Trying to find a faster algorithm is an open problem, especially for k = 3

Other operations

- The tree algorithm transfers immediately to the group ($\mathbf{Z}/2^n\mathbf{Z}$, +) (addition modulo 2^n)
- $-L = \{-x \bmod 2^n : x \in L\}$
- $L_{12} = L_1 \bowtie_l - L_2$
- $L_{34} = L_3 \bowtie_l - L_4$
- Solution is $L_{12} \bowtie - L_{34}$

Other operations

- Can apply to many other groups by defining joins correctly
- Can even apply to some non-group operations with special conditions

Finding many solutions

- For k = 4, we can find α^3 solutions with about α times the work of a single solution
- Take list size $(\alpha)(2^{n/3})$, then filter $(n/3 + \lg \alpha)$ bits at the low level of the tree
- $\alpha \leq 2^{n/6}$ must be true
- Generalizes to larger k

Lower bounds

- Information-theoretic
  - Computational complexity is $\Omega(2^{n/k})$
  - Gap between this and $O(k \bullet 2^{n/(1+\lg k)})$
- Discrete logs
  - If the discrete log problem in the group is hard, there is no polynomial-time algorithm for k-sum
  - If the discrete log problem in the group is easy, there is are nontrivial algorithms for k-sum

Uses for k-sum

$$H(x) \stackrel{\text{def}}{=} \sum_{i=1}^{k} h(i, x_i) \bmod 2^{256}.$$

- NASD incremental hash proposal
- Inverting the hash is a k-sum problem over the group ($\mathbf{Z}/2^{256}\,\mathbf{Z}$, +)

Inverting NASD hash

- Generate k lists L_1, ..., L_k
  - Pick many x values at random
  - In L_i, $y_i = h(i, x)$ for each x
- Solve $y^1 + ... + y^k \equiv c \pmod{2^{256}}$
- Any solution is a preimage for c
- k = 128, 128-sum yields a 128-block message with 2^{40} work

AdHash

$$H(x) \stackrel{\text{def}}{=} \sum_{i=1}^{k} h(i, x_i) \bmod m,$$

- Generalized NASD
- What value for m?
- Same attack on NASD works here
- Take k = $2^{\text{sqrt(lg m) - 1}}$, the complexity of the problem is $O(2^{2\ \text{sqrt(lg m)}})$
- For 80-bit security, m > 2^{1600}

PCIHF hash

$$H(x) \stackrel{\text{def}}{=} \sum_{i=1}^{n-1} \mathrm{SHA}(x_i, x_{i+1}) \bmod 2^{160} + 1.$$

- Changing x_i affects two terms
- Fix every other block
  - $x^2 = x^4 = x^6 = \ldots = 0$
  - Vary the other blocks
- This becomes:

$$H(x) = \sum_{j=1}^{\lfloor (n+1)/2 \rfloor} h(x_{2j-1}) \bmod 2^{160} + 1 \quad \text{where } h(w) \stackrel{\text{def}}{=} \mathrm{SHA}(0, w) + \mathrm{SHA}(w, 0).$$

- And we can run the NASD attack

PCIHF hash

- This computes preimages of H
  - n = 255, 255-block preimage with $O(2^{28})$ work
- We can extend the attack to find collisions by finding messages that hash to the same digest
- Also $O(2^{28})$ work

Open problems

- Other values of k
- Other combining operations other than xor and modulo addition
- Golden solution (or finding all solutions)
- Improved memory requirements and improved parallelization
- Lower bounds