Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities

Walter Cai
University of Washington
Seattle, Washington
walter@cs.washington.edu

Magdalena Balazinska
University of Washington
Seattle, Washington
magda@cs.washington.edu

Dan Suciu
University of Washington
Seattle, Washington
suciu@cs.washington.edu

ABSTRACT

In this work we introduce a novel approach to the problem of cardinality estimation over multijoin queries. Our approach leveraging randomized hashing and data sketching to tighten these bounds beyond the current state of the art. We demonstrate that the bounds can be injected directly into the cost based query optimizer framework enabling it to avoid expensive physical join plans. We outline our base data structures and methodology, and how these bounds may be introduced to the optimizer's parameterized cost function as a new statistic for physical join plan selection. We demonstrate a complex tradeoff space between the tightness of our bounds and the size and complexity of our data structures. This space is not always monotonic as one might expect. In order combat this non-monotonicity, we introduce a partition budgeting scheme that guarantees monotonic behavior. We evaluate our methods on GooglePlus community graphs [11], and the Join Order Benchmark (JOB) [16]. In the presence of foreign key indexes, we demonstrate a 1.7× improvement in aggregate (time summed over all queries in benchmark) physical query plan runtime compared to plans chosen by Postgres using the default cardinality estimation methods. When foreign key indexes are absent, this advantage improves to over 10×.

ACM Reference Format:
Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319894

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319894

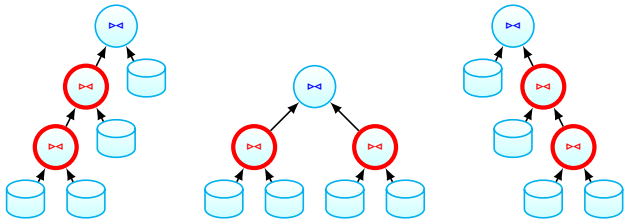


Figure 1: Join tree illustration where nodes represent relations. Intermediate relations highlighted in red.

1 INTRODUCTION

Cost based query optimizers use complex parameterized formulas to estimate the cost of candidate physical join plans. When presented with a multijoin query, most systems still rely on binary join algorithms to iteratively join these tables one-by-one into one or several growing intermediate relations. Figure 1 highlights these intermediate relations in red. A fundamental parameter to the formulas is the estimated size of these intermediate relations. If accurate row counts are available, there formulas generally avoid poor plans. However, when estimates are poor, the plan selection process becomes unreliable. Most optimizers have sophisticated methods of estimating selectivity of filter predicates applied to a single relation, but fall back on strong assumptions about the underlying data to handle join predicates. In this paper we focus on multijoin queries and the risks of intermediate relation blowup. Whereas key-foreign-key (K-FK) joins cannot exceed the size of the foreign key relation, queries threaten blowup when FK columns in two relations are joined. In the FK-FK join scenario, the resulting intermediate relation will often be many times larger –or even many degrees of magnitude larger– than the base relations.

Most production systems keep summary data (i.e. histograms, samples, other statistics) on base relations and use these to estimate the size of joins directly involving base relations, i.e. nodes whose two children are both base relations. However, it is difficult to reason about the size of final or intermediate relations when one or both of the child relations is itself an intermediate relation. Unfortunately,

the query optimizer will necessarily encounter this challenge for any query touching more than two tables. A naive solution is to collect statistics on a comprehensive list of intermediate products. This is highly impractical, especially if one considers all possible combinations of filter predicates placed on the base relations as these filters will, in turn, alter the intermediate relation. Instead, optimizers fall back on strong assumptions about the underlying data. In particular, many optimizers assume independence across joining columns in separate relations. Independence would suggest that common attribute values in one relation are not more likely to be common in the other relation. However, in real world datasets attribute value frequencies between joining columns are often highly correlated. This correlation renders the independence assumption invalid and leads to underestimation [17]. While overestimation can lead to marginally slower query plans with smaller than predicted intermediate joins, underestimation can lead to massive blowup during execution. Therefore, assuming independence is a high risk strategy.

As an alternative, we propose deriving size bounds for intermediate relations and allowing the optimizer to reason with respect to these bounds. Our contribution consists of a new technique that improves on known theoretical upper bounds, making them tighter and practical, by combining the theoretical upper bounds with sketches [8]. For each input relation we compute a sketch consisting of a small number of partitions. For each partition we compute the count (number of tuples), and the maximum degree with respect to each relevant join attribute from the query. The max degree of an attribute (e.g. the foreign key) is the largest number of occurrences of any value for that attribute in the table. To estimate the size of a multi-join query, we apply the theoretical upper bound formula to each combination of partitions of the relations and add their results. Thus, we replace the global statistics used by the known theoretical upper bound, with more granular statistics. Our estimate continues to be a theoretical upper bound on the true size of the query, yet is significantly more accurate than a naive application of the base theoretical formula.

For the case of a FK-FK join, degree information can significantly improve the estimate of the join. However, the theoretical upper bound formulas that use degrees are complex and require solving a linear optimization problem whose size is exponential in the size of the query [14]. Instead, we focus on a subset of all formulas which is simpler to compute and which performs well in practice. The practical subset is based on generating valid *coverages* of the join attributes using relations. Details are given in Section 3.5. We validate our technique using two broad metrics: tightening of bounds over the naive upper bound and query execution time. We measure execution time improvement by comparing the execution time of plans generated using bounds versus baseline unmodified Postgres optimization methods. Our results demonstrate that our partitioning strategy greatly tightens the bounds generated by the bounding formulas. Moreover, simply replacing conventional cardinality estimates with our bounds leads to overall faster physical plans.

We summarize the primary contributions of this paper as follows:

- We develop a novel approach to tightening theoretically guaranteed join cardinality upper bounds using random hash partitioning.
- We introduce an efficient and simple method of generating a practical subset of a complex space of entropic bounding formulas.
- We implement a partition budgeting strategy to control the space complexity of our sketches, and the time complexity of our bound calculation.
- We demonstrate the practicality of our approach on multiple real world datasets.

The remainder of the paper is organized as follows. In Section 2, we describe the theoretical foundations of our bounds. In Section 3, we introduce our data structure and approach. In Section 4, we evaluate our approach on two real world datasets using two metrics: bound tightness and true query execution time. Subsection 4.1 explores the tightness of our bounds, and Subsection 4.2 explores our execution time improvements on the JOB. We demonstrate a nearly 2× to 10× improvement in query plan execution time simply by using our bounds. In Section 5, we review the current state-of-the-art techniques in join cardinality estimation and other recent work in query optimization. Finally, in Section 6, we review our contributions and describe future directions for this line of research. We also include a symbol table for reference in Table 1.

2 BACKGROUND

In the traditional query execution pipeline, the query optimizer accepts a logical representation of a query over a database instance and returns one of potentially millions of physical plans to execute. The binary join algorithm paradigm generally forces physical plans to take a tree-structure where intermediate relations are incrementally produced during execution. Cardinality estimation refers to the estimates a query optimizer generates to predict the size of intermediate joins in a physical join tree.

A combination of data summarization techniques (e.g. multidimensional histograms), hand-written rules, and strong assumptions on the underlying data remain the de facto estimation method for production systems [22, 25]. For longer

Notation	Definition
D	database instance
R, R_j, S, T	relations
t	tuple
a, a_i	attributes
$\boldsymbol{a}$	attribute sets
$\boldsymbol{a}_j$	attributes of relation R_j
Q	query head
$\mathcal{H}$	hypergraph
$(u_1, \ldots, u_m)$	fractional edge cover
H	hash function
W	domain of attributes
$[M]$	the set $\{1, \ldots, M\}$
I	hash value array where entries correspond to attributes $a \in \boldsymbol{a}$: $I \in [M]^{\lvert\boldsymbol{a}\rvert}$
T^I, R^I, R_j^I	relation subsets corresponding to hash value array I
D^I	subset of database instance corresponding to hash value array I
c, c_R, c_{R^I}	count statistic
$d^a, d_R^a, d_{R^I}^a$	max degree statistic with respect to attribute a
B	partition budget
h	entropy
$\mathcal{S}, \mathcal{S}_R$	relation sample

Table 1: Symbol table.

multijoins or cyclic joins, these approaches becomes increasingly inaccurate [16] and in particular gravitate towards underestimation in real world settings. Underestimation leads to poor physical plan selection, which in turn leads to poor performance. We argue that providing guaranteed bounds on intermediate join cardinality can make selection more robust.

The first step towards these bounds comes from analyzing the connection between information theory and relational databases. Framing relational joins as hypergraphs, Atserias et al. generate upper bounds based on fractional edge covers [4]. More formally, take a conjunctive query on relations $R_1, \ldots, R_m$ with attribute collections $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m$.

$$Q\left(\boldsymbol{a}\right) \text{:-} R_1\left(\boldsymbol{a}_1\right), \ldots, R_m\left(\boldsymbol{a}_m\right)$$

where $\boldsymbol{a} = \cup_j \boldsymbol{a}_j$. We define a *hypergraph* $\mathcal{H}$ which models the schema of the query. A hypergraph is a generalization of an undirected graph where the *hyperedges* of the hypergraph are defined as arbitrary nonempty subsets of the vertex set instead of only strictly size 2 subsets. Let the vertices of $\mathcal{H}$ corresponding to the attributes $a \in \boldsymbol{a}$. For each relation R_j include a hyperedge corresponding to the attributes-set $\boldsymbol{a}_j$. For the query given in Figure 2a, we provide the corresponding hypergraph in Figure 2b. A *fractional edge cover* $(u_1, \ldots, u_m)$ of $\boldsymbol{a}$ on $\mathcal{H}$ is a set of values $u_j \in \mathbb{R}_{\geq 0}$ corresponding to the hyperedges $\boldsymbol{a}_j$ where for all vertices a of the hypergraph, the sum of the u_j values corresponding to hyperedges containing a is at least 1. That is, attribute a is *covered*:

$$\forall a \in \boldsymbol{a} : \quad \sum_{j : a \in \boldsymbol{a}_j} u_j \geq 1$$

We may bound the join cardinality as follows:

$$|Q| \leq \prod_{j=1}^{m} \left|R_j\right|^{u_j}$$

This class of bounds is referred to as the *AGM bound* after being originally developed by Atserias et al [4]. A proof depends on Shearer's Lemma and may be found in Appendix A. Note that these formulas assume uniqueness of full tuples within each relation. This restriction is a fact that must be enforced by the underlying database system. Note that while some bounding formulas are still valid even in the presence of repeated rows, this is not generally the case.

Khamis et al. extend the AGM bound to include degree parameters [14]. Their contributions generalize the information theory versus relational join analogy to include conditional entropic formulations. We refer to this broader class of bounds as the *KNS bound*. While the KNS bound is broader, the actual calculation of all formulas within KNS can be highly complex. For this reason, we will demonstrate how to produce a pared down but still effective subset in Subsection 3.5.

3 APPROACH

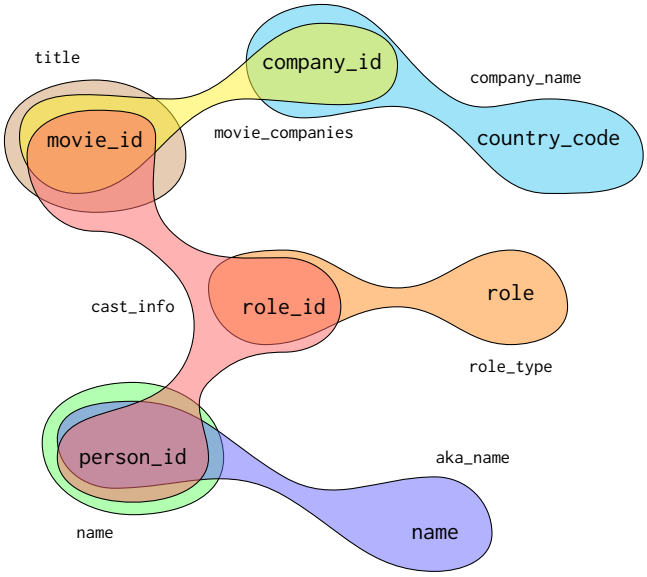
In this Section we present our approach. The key idea is to use data sketches to tighten the bounds generated in Section 2. The remainder of this Section is divided up as follows. In Section 3.1, we define our core data structure; the *Bound Sketch* (BS). In Section 3.2, we describe how the BS is used to generate and tighten theoretical join cardinality upper bounds. We followup this discussion with two performance optimizations which make bounding more feasible. In Section 3.3, we describe our selection predicate propogation technique as well as the reasoning behind using it. In Section 3.4, we introduce our hash bucketization budgeting scheme. We provide an illustration of our full workflow in Figure 3. Finally, in Section 3.5, we describe the process of automatically generating the bounding formula given the query's hypergraph structure.

3.1 The Bound Sketch

We first describe the structure of the BS. Take a relation $T(\boldsymbol{a})$ with attributes $a \in \boldsymbol{a}$ and random hash function $H : W \mapsto \{1, \ldots, M\}$. Let $[M]$ denote $\{1, \ldots, M\}$ which is the image

```sql
SELECT
    MIN(a1.name) AS writer_pseudo_name,
    MIN(t.title) AS movie_title
FROM
    aka_name AS a1,
    cast_info AS ci,
    company_name AS cn,
    movie_companies AS mc,
    name AS n1,
    role_type AS rt,
    title AS t
WHERE
    cn.country_code = '[us]'
    AND rt.role = 'writer'
    AND a1.person_id = n1.id
    AND n1.id = ci.person_id
    AND ci.movie_id = t.id
    AND t.id = mc.movie_id
    AND mc.company_id = cn.id
    AND ci.role_id = rt.id
    AND a1.person_id = ci.person_id
    AND ci.movie_id = mc.movie_id;
```

(a) SQL query.



(b) Corresponding hypergraph $\mathcal{H}$.

Figure 2: JOB query 08c.

of the hash function. Tuples in T take values in the domain $W^{|\boldsymbol{a}|}$. For any tuple t, let $t[a]$ refer to the attribute value of a in t. Similarly, let the index $I \in [M]^{|\boldsymbol{a}|}$ be an array of hash values and let $I[a]$ indicate the hash value corresponding to attribute a. For each index value I, we define the subset T^I of relational instance T as those tuples $t \in T$ that hash to the values in I.

$$T^I = \Big\{ t \in T : H(t[a]) = I[a], \quad \forall a \in \boldsymbol{a} \Big\}$$

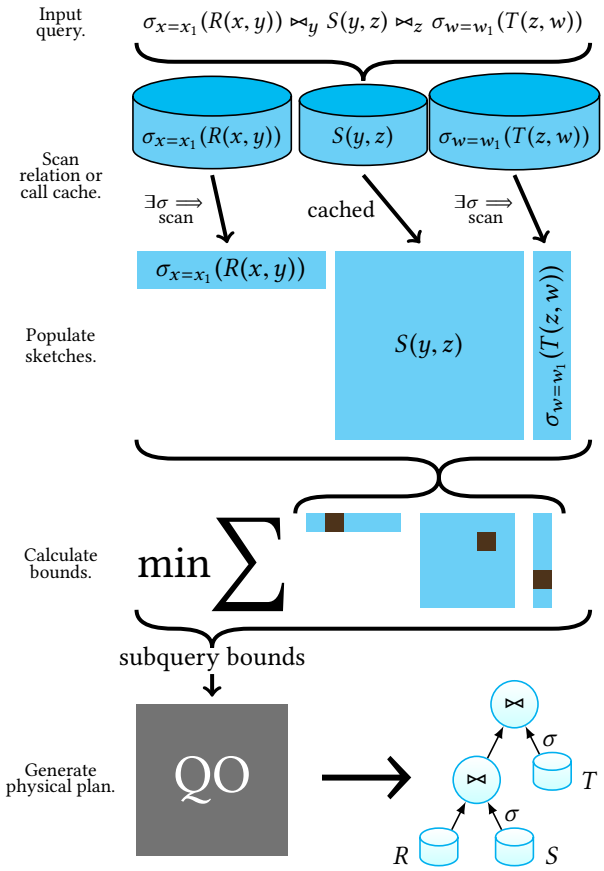


Figure 3: Illustration of the bounds workflow. Given a query, we either retrieve the necessary sketch from cache or scan the relation to populate a sketch reflecting the relevant filter predicates. For all relevant subqueries, we then calculate the cardinality bound and allow the cost-based query optimizer to reason based on these bounds.

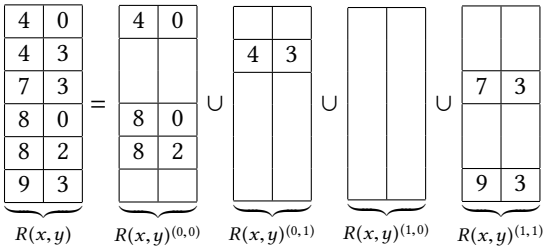
The BS on T is a collection of $|\boldsymbol{a}|+1$-many $|\boldsymbol{a}|$-dimensional tensors of the form $M \times \cdots \times M$ Each cell of the first tensor contains a *count* value c. We define the I-th entry of this tensor as $c_{T^I} = |T^I|$. Each cell of the remaining $|\boldsymbol{a}|$ tensors contains a *degree* value d. Each of these tensors corresponds to a conditional attribute in $\boldsymbol{a}$. We define the degree parameter corresponding to variable $a \in \boldsymbol{a}$ as the maximum degree for attribute a from the partition T^I. More formally:

$$d_{T^I}[a] = \max_{\substack{w \in W: \\ \{H(w)=I[a]\}}} \Big| \underbrace{\{t : t[a] = w\}}_{\subseteq T^I} \Big|$$

Finally, we describe the BS as:

$$\Big(\underbrace{[c_{T^I}]}_{\text{count tensor}} , \underbrace{[d_{T^I}[a_1]], \ldots, [d_{T^I}[a_{|\boldsymbol{a}|}]]}_{|\boldsymbol{a}|\text{-many degree tensors}} \Big) \in \left([M]^{|\boldsymbol{a}|} \right)^{|\boldsymbol{a}|+1}$$

As an illustrative example, consider the following relation $R(x, y)$. The hash function is an indicator for even values: $h(x) = x\%2$. We first logically partition R into $2 \cdot 2 = 4$ subsets:



This data generates the following tensor components in the BS:

$$\left(c_R = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix},\ d_R[x] = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix},\ d_R[y] = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix} \right)$$

We further note it is possible to create the BS from a single pass over the relation. For more details, we refer the reader to Appendix C.

3.2 Our Cardinality Bounds

Next, we describe how the BS is used to tighten guaranteed cardinality upper bounds compared to when applying the bounding formula without use of the BS. Consider schema

$$R_1(\boldsymbol{a}_1), \ldots, R_n(\boldsymbol{a}_n)$$

and database instance D on relations $R_1, \ldots, R_n$. Let $\boldsymbol{a} = \cup_j \boldsymbol{a}_j$ be the set of attributes. In this context if two attributes are equijoined in the given query, then they are considered the same attribute. Define index array $I \in [M]^{|\boldsymbol{a}|}$ as before, and let $I[\boldsymbol{a}_j]$ be the sub-array of I whose values correspond to the attributes present in $\boldsymbol{a}_j$. Define a database instance D^I as a subset of database instance D where for each relational instance $R_j(D)$, we take the subset

$$R_j\left(D^I\right) = \left\{ t \in R_j(D) : \forall a \in \boldsymbol{a}_j, \quad h\left(t\left[a\right]\right) = I[a] \right\}$$

That is, the set of all tuples in $R_j(D)$ which hash to the correct index values for each attribute in $\boldsymbol{a}_j$. Define database instance subset $D^I = \left\{ R_1\left(D^I\right), \ldots, R_m\left(D^I\right) \right\}$. Given a conjunctive query $Q(D)$, observe that we may reconstruct the full conjunctive query using only these D^I:

$$Q(D) = \bigcup_{I \in [M]^{|\boldsymbol{a}|}} Q\left(D^I\right) \tag{1}$$

Note that this is a necessarily disjoint union. We now invoke the primary results of the AGM and KNS bounds. As an illustrative example we describe a triangle query:

$$Q\left(x, y, z\right) \text{:-} R\left(x, y\right), S\left(y, z\right), T\left(z, x\right) \tag{2}$$

Take a triple of random variables (X, Y, Z) corresponding to attributes x, y, z, respectively, and ranging uniformly over the collection of all tuples in the true output. The size of the query is tied to the joint entropy of all three variables. Specifically, $h(X, Y, Z) = \log |Q(x, y, z)|$. By construction on the domain space of the triple (X, Y, Z), and since our query only includes equivalence join predicates, for any subset of $\{X, Y, Z\}$ which happens to correspond exactly to the attributes of some relation in our schema we may bound the entropy of that subset of variables. Specifically:

$$h(X, Y) \leq \log(c_R), \quad h(Y, Z) \leq \log(c_S), \quad h(Z, X) \leq \log(c_T)$$

Similarly, we may relate conditional entropic formulas to our degree statistics:

$$\begin{aligned} h(X|Y) &\leq \log(d_R^y), & h(Y|X) &\leq \log(d_R^x) \\ h(Y|Z) &\leq \log(d_S^z), & h(Z|Y) &\leq \log(d_S^y) \\ h(Z|X) &\leq \log(d_T^x), & h(X|Z) &\leq \log(d_T^z) \end{aligned}$$

We may exploit conditional subadditivity, as well Shearer's lemma (Lemma A.2) to generate entropic bounds for $h(X, Y, Z)$:

$$h(X, Y, Z) \tag{3}$$
$$\leq \begin{cases} h(X, Y) + h(Z|Y),\ h(Y, Z) + h(X|Z),\ h(Z, X) + h(Y|X) \\ h(X, Y) + h(Z|X),\ h(Y, Z) + h(X|Y),\ h(Z, X) + h(Y|Z) \\ \frac{1}{2}h(X, Y) + \frac{1}{2}h(Y, Z) + \frac{1}{2}h(Z, X) \end{cases}$$

Each of these entropic bounding expressions corresponds to a bounding formula for $Q(x, y, z)$. We enumerate the corresponding query cardinality bounding formulas for the entropic bounding expressions in Equation 3:

$$|Q(D)| \leq \begin{cases} c_R d_S^y, & c_S d_T^z, & c_T d_R^x \\ c_R d_T^x, & c_S d_R^y, & c_T d_S^z \\ (c_R)^{\frac{1}{2}} (c_S)^{\frac{1}{2}} (c_T)^{\frac{1}{2}} \end{cases} \tag{4}$$

While these formulas represent the complete KNS bound, we often will only generate a subset. Specifically, those formulas expressible with the BS and also not strictly dominated by other bounding formulas. For a full justification, we refer the reader to Appendix A as well as the original papers: [4, 14]. Finally, we may combine the bounds over the database partitions defined in Equation 1. Summing over all index

combinations provides an upper bound on D.[1]

$$|Q(D)| \leq \sum_{I \in [M]^{|\boldsymbol{x}|}} \min \begin{cases} c_{R^I} d_{S^I}^{y},\ c_{S^I} d_{T^I}^{z},\ c_{T^I} d_{R^I}^{x} \\ c_{R^I} d_{T^I}^{x},\ c_{S^I} d_{R^I}^{y},\ c_{T^I} d_{S^I}^{z} \\ (c_{R^I})^{\frac{1}{2}} (c_{S^I})^{\frac{1}{2}} (c_{T^I})^{\frac{1}{2}} \end{cases} \tag{5}$$

3.3 Selection Propagation and Preprocessing

We now describe an optimization to make calculation of these bounds more tractable. Specifically, we propagate selections through foreign key joins to simplify the join topology. We again consider the JOB query featured in Figure 2. We provide an alternative relation-centric graphical representation of this query in Figure 4. In this illustration the nodes represent relations and edges between the nodes represent join predicates. Note that there exist 4 distinct join attribute equivalence classes across the seven tables. Specifically, these are the ID's for individual films, people, companies, and role types represented by edge colors blue, red, orange, and violet, respectively. Primary-key-foreign-key (PK-FK) constraints are represented by arrows pointing in the direction of the relation where the attribute is a primary key. Dashed lines represent foreign-key-foreign-key (FK-FK) joins. To increase efficiency at the planning stage, it is best to eliminate as many join attributes as early as possible.

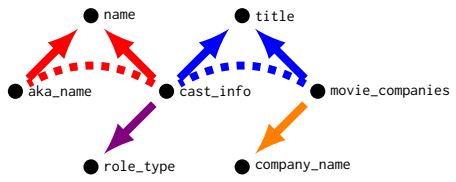


Figure 4: JOB query `08c` join attribute graph.

This is easily done by first analyzing the query topology and then eliminating join attributes (and hence also some relations) which will not increase the output cardinality. For instance, JOB query `08c` features the selection:

```
role_type.role = 'writer'
```

This particular string selection corresponds to a set of role type ID's. Often this is a highly selective predicate and in this case only corresponds to a single ID: {4}. It is therefore possible to preprocess the relation `cast_info` and generate a fresh BS on $\sigma_{\texttt{role\_type\_id=4}}(\texttt{cast\_info})$.

This is not practical (or beneficial) in all cases. For instance, JOB query `08c` also features the following selection:

```
company_name.country_code = '[us]'
```

[1] We observe increasing our hash size generally tightens this bound although not in all cases. See Appendix B.

In contrast to the previous example, this selection will correspond to several thousand company ID's[2]. Our system detects the large number of returned IDs while preprocessing `company_name` and will instead default to generating an updated BS for

$$\sigma_{\texttt{country\_code='[us]'}}(\texttt{company\_name})$$

In this manner, we decompose a join into only those predicates which threaten to cause blowup in intermediate products.

Preprocessing also helps us broaden the scope of our method. Although we are currently only suited to handle equivalence predicates during our bound generation, propagation of selections allows us to handle `LIKE`, and inequality ($\geq, >, \neq, \leq, <$) filter predicates. This method requires scanning relevant base relations at runtime. However, only a single scan is required and we argue that for queries featuring FK-FK joins between large relations, the preprocessing time is insignificant compared to the risk of poor plan selection. In the case of our running example (JOB `08c`), these "dangerous" attributes featuring FK-FK join predicates are people, and films and are highlighted as dashed lines in Figure 4.

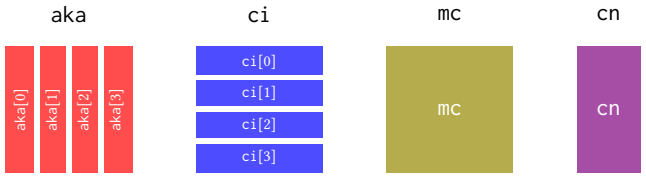
We emphasize that some sketches can be cached ahead of time, while others must be populated at runtime. If a relation is subject to any filter predicates following filter selection propogation, then the BS associated with that relation must be populated at runtime. Otherwise, the BS may be drawn from cache and will not require a scan of the table.

3.4 Hash Partition Budgeting

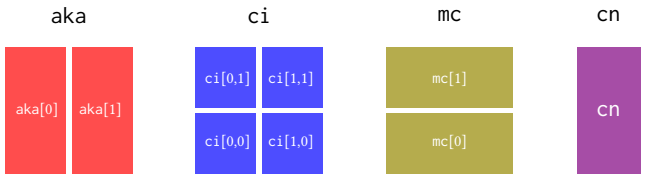
The hash partitioning approach on join attributes leads to the question: how many buckets should be used for each join attribute? If we set a fixed size for each attribute, depending on the number of join attributes appearing in the query, the set of all hash value combinations can grow exponentially with the number of buckets even after selection propagation. Generating a bound for each combination of hash values can quickly become computationally impractical as the number of join attributes grows. Furthermore, it is in fact possible for the sum of these bounds to increase as we increase hash size. This is due to hash collisions between non-joining tuples, and the exponential in hash size number of terms (hash value combinations) in the bound summation. We have included a simple example in Appendix B demonstrating the potentially non-monotonic behavior of our bound.

We simultaneously address both these problems by introducing a hash partition budgeting scheme. The underlying idea is simple: for any bounding formula, and bucket threshold B, we only calculate a bound for at most B hash value combinations. We distribute this budget to those attributes

[2] 132,917 individual companies



(a) Partitioning for formula $c_{\texttt{aka}} \cdot d^{y}_{\texttt{ci}} \cdot d^{z}_{\texttt{mc}}$, budget $B = 4$.



(b) Partitioning for formula $d^{y}_{\texttt{aka}} \cdot c_{\texttt{ci}} \cdot d^{z}_{\texttt{mc}}$, budget $B = 4$.

Figure 5: Hash partition budgeting illustration for formulas in Equations 7 and 8 regarding the query in Equation 6.

that are covered *unconditionally* by a relation. That is, the variable associated with the attribute appears in an unconditional entropic term in the entropic bounding formula associated with the cardinality bounding formula. For example, consider the following query:

$$Q(x, y, z, w) \text{:-} \texttt{aka}(x, y), \texttt{ci}(y, z), \texttt{mc}(z, w), \texttt{cn}(w) \tag{6}$$

This query yields multiple bounding formulas including the following two examples and their corresponding entropic formulas:

$$c_{\texttt{aka}} \cdot d^{y}_{\texttt{ci}} \cdot d^{z}_{\texttt{mc}} : \qquad h(X, Y) + h(Z|Y) + h(W|Z) \tag{7}$$

$$d^{y}_{\texttt{aka}} \cdot c_{\texttt{ci}} \cdot d^{z}_{\texttt{mc}} : \qquad h(X|Y) + h(Y, Z) + h(W|Z) \tag{8}$$

For the bound formula in Equation 7, we dedicate all of our B buckets to the only unconditionally covered join attribute: y (covered unconditionally by aka). Note that x is not a join attribute; we would not benefit from partitioning x. In this scenario, the number of hash buckets for y, z, and w are B, 1, and 1, respectively. That is, z and w are not partitioned at all. Alternatively, for the bound formula in Equation 8, we observe that join attributes y, and z are both covered unconditionally by ci. We therefore dedicate $\sqrt{B}$ buckets to both y and z so that the total number of hash-index combinations is $\sqrt{B} \cdot \sqrt{B} = B$. We provide illustrations for the bounding formulas in Equations 7, and 8 in Figures 5a, and 5a, respectively where $B = 4$.

By only partitioning unconditionally covered variables, we get the desired behavior that increasing partition budget should tighten our bound. In fact, if we only increase the partitioning budget by integer factors from B to $B' = B \cdot r$ and thereby only allow subpartitioning of existing buckets, this monotonic behavior is guaranteed. Nonmonotonic behavior is not guaranteed when partitioning join attributes that are covered conditionally.

The primary caveat is that the index combinations are no longer directly comparable between distinct bounding formulas since each formula now demands a different hashing scheme. The general bounding expression now minimizes over all bounding formulas instead of over all index combinations. While this does remove some inherent gains from fine granularity minimization (i.e. minimizing inside Equation 5's summation), our experiments demonstrate that for a single query, a majority of index combinations tend to favor a single bounding formula. Hence, this modification represents a worthy trade off between the benefits of fine grained comparison, robust performance, and sufficient generality to include queries with a large number of join attributes. Our new general expression is:

$$|Q(D)| \leq \min_{\substack{b \in \\ \text{bounding formulas}}} \left(\sum_{\substack{I \in \\ \text{partition indexes}}} b\left(Q\left(D^{I}\right)\right) \right) \tag{9}$$

This modification also affects the sketch generation step. We observe that even for a single partitioning budget, different bounding formulas might each necessitate a BS from the same relation but of different dimension. However, since relations will generally not exceed three foreign keys, there are few combinations of relevant tensor dimensions leading to few sketches describing the same relation. Furthermore, each sketch is inherently limited to the budgeted number of partitions B (or is many degrees of magnitude smaller) and hence the storage overhead for storing offline sketches is also low.

As an illustrative example, consider again the query in Equation 6 which we reproduce below for convenience:

$$Q(x, y, z, w) \text{:-} \texttt{aka}(x, y), \texttt{ci}(y, z), \texttt{mc}(z, w), \texttt{cn}(w)$$

The bounding formulas $d^{y}_{\texttt{aka}} \cdot c_{\texttt{ci}} \cdot d^{z}_{\texttt{mc}}$ will require a $\sqrt{B} \times \sqrt{B}$ dimension BS for relation ci, whereas bounding formula $c_{\texttt{aka}} \cdot d^{y}_{\texttt{ci}} \cdot d^{z}_{\texttt{mc}}$ will require a $B \times 1$ dimension BS (with respect to attribute y). However, since both sketches may be generated in a single pass, we may construct them in parallel.

3.5 Bound Formula Generation

We continue with our running example of JOB query 08c. Selection propagation helps us eliminate one of four join attributes. In this task, we employ a further optimization: ignoring remaining dangling K-FK joins. We do this because the degree multiplier on the key side of the join is one (by definition). It is usually not the case that this information will help to tighten the bound. For example, when considering the join movie_companies $\bowtie$ title, the resulting output will be determined by the FK side of the join,

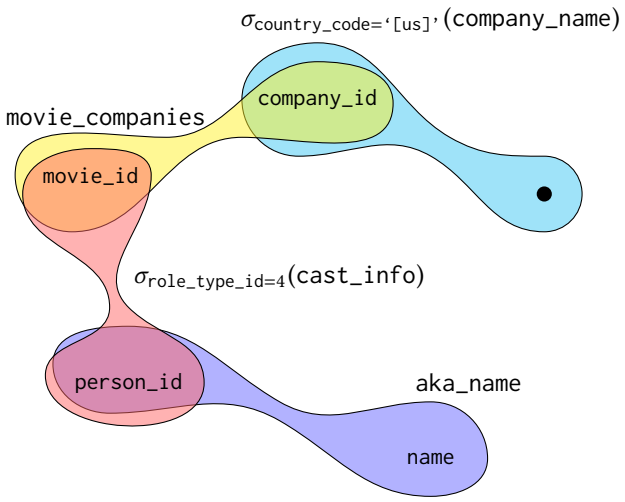


Figure 6: Hypergraph of JOB `08c` following propagation of the `role_type` and `country_code` selection predicates.

in this case `movie_companies`. In fact, this is always true assuming the database does not allow dangling FK pointers to nonexistent keys. We therefore ignore joins satisfying this restriction. Note that in JOB query `08c`, the inclusion of the relation `company_name` does not satisfy these conditions because there is a further selection predicate on `company_name.country_code`. While this selection is not sufficiently selective to warrant full selection propagation, it is significant enough to warrant the presence of `company_name` in our bounding formulas. The selection predicate has created a situation where keeping a dangling FK pointer in the `movie_companies` relation might be beneficial. The modified hypergraph following selection propagation and removal of (most of the) dangling K-FK joins may be found in Figure 6. Note that we left an unnamed "attribute" represented simply by a dot in the the

$$\sigma_{\texttt{country\_code='[us]'}}(\texttt{company\_name})$$

relation. This is because `company_id` is a foreign key in `company_name` but not a primary key. This inherently implies there exists other attributes which `company_name` must cover and avoids violating the duplicate tuples restriction.

The task of generating bounding formulas is as follows. We first consider all combinations of attribute coverages and filter those which do not fit within our bounding scheme. Using these coverages, we then build our explicit bounds. The pseudocode for generating our bounding formulas and coverages may be found in Algorithms 1 and 2.

The algorithm ingests the attribute-centric hypergraph representation and iterates through coverage assignments for each attribute. That is, for every coverage combination,

Algorithm 1 Bound Formula Generator. Given a hypergraph representation of a query, returns a collection of bounding formulas where each formula takes the form of a 2-tuple: $(c, d = (r, a)) \subseteq R \times (R \times A)$. We let c be the set of relations contributing count statistics. We let $d = (r, a)$ be the set of pairs of relations r and corresponding attributes a, where r contributes a degree statistics with respect to a.

procedure BOUNDING_FORMULAS($\mathcal{H} = (A, R)$) ▷ input hypergraph where A are attributes (vertices) and R are Relations (hyperedges)
  $\mathcal{B} \leftarrow$ Gen_Covers($\mathcal{H}$) ▷ generate coverages
  $\mathcal{F} \leftarrow \{\}$ ▷ bounding formulas
  for $\boldsymbol{r} = (r_1, \ldots, r_{|A|}) \in \mathcal{B}$ **do**
    $c \leftarrow \{\}$ ▷ relations contributing count terms
    $d \leftarrow \{\}$ ▷ relations contributing degree terms
    for $r \in R$ **do**
      cover_count $\leftarrow |\{r' \in \boldsymbol{r} : r' = r\}|$
      if cover_count $= 0$ **then**
        pass ▷ relation r does not appear in bounding formula
      else if cover_count $= |r|$ **then**
        $c \leftarrow c \cup \{r\}$
      else if cover_count $= |r| - 1$ **then**
        a ▷ a will be unique attribute in r that is not paired [covered] by r in $\boldsymbol{r}$
        for $a' \in r$ **do**
          if $\boldsymbol{r}[a] \neq r$ **then**
            $a = a'$
        $d \leftarrow d \cup \{(r, a)\}$
      else ▷ all other possibilities filtered by Gen_Covers
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{(c, d)\}$
  return $\mathcal{F}$ ▷ bounding formulas

Algorithm 2 Feasible Coverage Generator. Given a hypergraph representation of a query, returns a restricted collection of attribute to covering relation mappings.

procedure GEN_COVERS($\mathcal{H} = (A, R)$) ▷ input hypergraph
  $C \leftarrow \prod_{a \in A} \{r \in R : a \text{ covered by } r\}$ ▷ cross product
  $\mathcal{B} \leftarrow \emptyset$
  for $\boldsymbol{r} = (r_1, \ldots, r_{|A|}) \in C$ **do**
    `safe = True`
    for $r \in R$ **do**
      if $|\{r' \in \boldsymbol{r} : r' = r\}| \notin \{0, |r|, |r| - 1\}$ **then**
        `safe = False`
    if `safe` **then**
      $\mathcal{B} \leftarrow \mathcal{B} \cup \{\boldsymbol{r}\}$
  return $\mathcal{B}$ ▷ feasible coverages

each attribute will be covered by precisely one of the relations which features that attribute. For every possible coverage combination we consider only those that are expressible by elements of the BS. This implies for all relational hyperedges e, the number of attributes covered by e must take one of three possible values in order for our sketches to be applicable: 0, $|e|$, or $|e| - 1$.

The intuition for these three possibilities is as follows:

- If a relation has coverage size 0, then it has no coverage responsibilities and need not appear in the resulting bound formula at all.
- If a relation has coverage size equal to the number of attributes present in the relation, then this corresponds to an unconditional entropic term. Therefore, the count term for the relation will appear in the resulting bound formula.
- If a relation has coverage size equal to the number of attributes present in the relation minus 1, then this corresponds to a conditional entropic term conditioned on that missing attribute. Therefore, the degree term for that relation with respect to that missing attribute will appear in the resulting bound formula.

We note that other coverages can also lead to valid bounding formulas. In case a join attribute is covered by more than one relation, the bound will still be valid, but the formula will be dominated by a similar formula where that attribute is instead only covered by a single relation. That is, there exists a formula that must produce a bound that is less than or equal to the multiple coverage bound. Another consideration is those coverages where there exists a relation e which covers at least one attribute, but fewer than the number of join attributes present in e minus 1. This corresponds to a degree term with respect to greater than one attribute. Since we restrict our BS's to not include these terms, we cannot use these coverages. The generated coverages and corresponding bound formulas for JOB query `08c` are enumerated in Figure 7. For readability, we refer to relations with their name abbreviations as it appears in Figure 2a. We abuse notation and refer to the filtered relation $\sigma_{\texttt{role\_type\_id=4}}(\texttt{cast\_info})$ simply as `ci`.

Note that some bounding formulas entirely separate the query graph into disjoint subgraphs (formulas 2, 3, 4, and 6). On the other hand, some formulas treat the join as a single long chain where they start with the count term on a single relation and build outward with degree multipliers (formulas 1, 5, 7, and 8). Finally, note that only formulas involving the count term $c_{\texttt{cn}}$ on `company_name` (formulas 2, 4, 6, and 8) are likely to benefit from the selection predicate on `company_name.country_code`. This is because `companyID` is a key in `company_name` but the selection is not highly selective. Hence, the remaining formulas will instead feature

	name	peopleID	filmID	companyID	·
1	a1	a1	ci	mc	cn
2	a1	a1	ci	cn	cn
3	a1	a1	mc	mc	cn
4	a1	a1	mc	cn	cn
5	a1	ci	ci	mc	cn
6	a1	ci	ci	cn	cn
7	a1	ci	mc	mc	cn
8	a1	ci	mc	cn	cn

(a) Coverage combinations.

	formula
1	$c_{\texttt{a1}} \cdot d_{\texttt{ci}}^{\texttt{peopleID}} \cdot d_{\texttt{mc}}^{\texttt{filmID}} \cdot d_{\texttt{cn}}^{\texttt{companyID}}$
2	$c_{\texttt{a1}} \cdot d_{\texttt{ci}}^{\texttt{peopleID}} \cdot c_{\texttt{cn}}$
3	$c_{\texttt{a1}} \cdot c_{\texttt{mc}} \cdot d_{\texttt{cn}}^{\texttt{companyID}}$
4	$c_{\texttt{a1}} \cdot d_{\texttt{mc}}^{\texttt{companyID}} \cdot c_{\texttt{cn}}$
5	$d_{\texttt{a1}}^{\texttt{peopleID}} \cdot c_{\texttt{ci}} \cdot d_{\texttt{mc}}^{\texttt{filmID}} \cdot d_{\texttt{cn}}^{\texttt{companyID}}$
6	$d_{\texttt{a1}}^{\texttt{peopleID}} \cdot c_{\texttt{ci}} \cdot c_{\texttt{cn}}$
7	$d_{\texttt{a1}}^{\texttt{peopleID}} \cdot d_{\texttt{ci}}^{\texttt{filmID}} \cdot c_{\texttt{mc}} \cdot d_{\texttt{cn}}^{\texttt{companyID}}$
8	$d_{\texttt{a1}}^{\texttt{peopleID}} \cdot d_{\texttt{ci}}^{\texttt{filmID}} \cdot d_{\texttt{mc}}^{\texttt{companyID}} \cdot c_{\texttt{cn}}$

(b) Bound formulas.

Figure 7: JOB query `08c` coverage combinations and their corresponding bound formulae.

$d_{\texttt{cn}}^{\texttt{companyID}}$ terms which will almost always take value one during actual calculation.

4 EVALUATION

We evaluate our bounds on two real world datasets. The first is a collection of 45 GooglePlus community edge-sets [11]. The edge counts within the communities range between 228,521 and 1,614,977. Each tuple (A, B) in an edgeset represents an "*A follows B*" relationship and is therefore not necessarily bidirectional. The edgesets therefore comprise directed graphs. We use the JOB for our plan execution time evaluation. The JOB is based on the IMDb dataset [1] and features several entity table as well as association tables relating entity tables to one another. The largest relation in the IMDb dataset is `cast_info` which assigns cast members to specific films and contains 59,906,495 rows. All experiments are run on a modified Postgres 9.6.6 instance. Note the only modification is to allow the optimizer to substitute our bounds from an external module and the execution engine remains unchanged. We use the non-cryptographic Murmur3 algorithm modulo the hash size with varying seed values [2] as our hash function. We highlight that our proposed modification is typically a lightweight change leaving almost all of the remaining database engine untouched.

4.1 Progressive Bound Tightness

We demonstrate that increasing the partitioning budget may significantly tightens our bounds. With this in mind, we design a microbenchmark based on GooglePlus community edgesets. We construct 11 distinct query templates, each featuring a unique topology and involving between two and five relations. Examples are give in Appendix D. Each template yields 20 distinct queries based on a different GooglePlus edgeset. The specific community that is chosen for each query is chosen uniformly and without replacement from the collection of 45. Furthermore, each template includes random filter predicates on different relations throughout each query. The filter predicates take the form

$$\text{table.follower\_id} \ \% \ K = x$$
$$\text{table.followed\_id} \ \% \ K = x$$

K is tuned for each template in order to control the relative size of each query output. For instance, queries with more relations tend to generate a larger result due to the presence of more FK-FK join attributes. In this situation, K is higher resulting in a more selective filter. K is smaller for templates with fewer relations. x is chosen at random for each individual query, and independently from each filter within the same query. Note that the microbenchmark is comprised of self-join queries analogous to the subgraph isomorphism count problem within a single GooglePlus community [10]. While joining across different communities is generally possible, the joining columns are often insufficiently correlated to capture a significant FK-FK join blowup. Hence, we rely on self-joins.

We execute the microbenchmark for partition budgets 1, 8, 64, 512, and 4096. For comparison, we also include the results using Postgres' default query optimizer. Our metric is Relative Error (RE) which we define as follows:

$$\text{RE}(\text{truth}, \text{estimate}) = \begin{cases} \infty & \text{truth} = 0 \ \& \ \text{estimate} > 0 \\ 1.0 & \text{truth} = 0 \ \& \ \text{estimate} = 0 \\ \frac{\text{estimate}}{\text{truth}} & \text{else} \end{cases}$$

Figure 8 includes histograms of the RE between the true cardinality, and the estimate or bound of each materialized join and subjoin. Each histogram operates on a logarithmic bucket scale on the x-axis. We observe that as the partitioning budget grows, the observed bounds shrink dramatically. At budget 4096, the majority of estimates hover just above the true value indicating it is likely impractical to increase budget any further. We also note that Postgres does not strongly favor underestimation in these experiments. This is because the queries from the microbenchmark include filters on arbitrary IDs that do not pertain to the real world. They therefore do not introduce the inherent correlation one might expect from a community graph. Note that Postgres' estimates cast a much wider logarithmic RE distribution than using bounds. This suggests the relative RE between different subqueries can be significantly larger and can lead to poor join orderings.

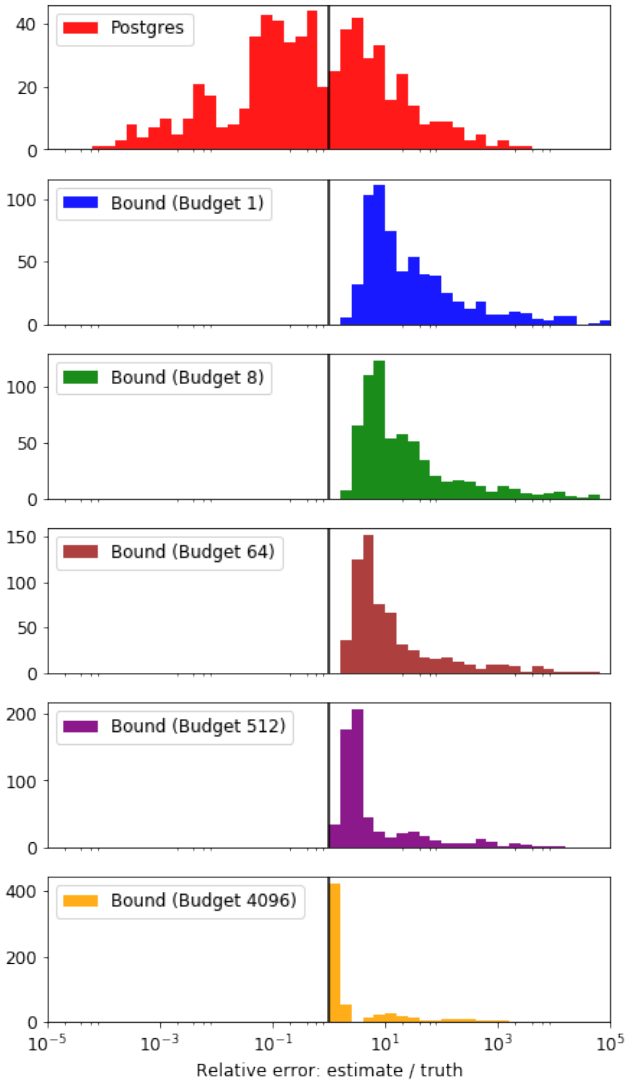


Figure 8: Relative Error of the default Postgres optimizer (labeled *Postgres*), and of our bounds for an increasing hash partitioning budget over the GooglePlus microbenchmark. Using bounds with partitioning budget one corresponds to the traditional KNS bound.

Finally, we demonstrate using these bounds translates into plan execution time improvement. For each of the above partitioning budgets and default Postgres, we include an aggregate average plan execution time. This value is the sum of the average plan execution time of five runs for each

Optimizer Setting	Aggregate Plan Runtime
Default Postgres	46.9 minutes
Budget 1	39.1 minutes
Budget 8	39.1 minutes
Budget 64	37.2 minutes
Budget 512	36.9 minutes
Budget 4096	36.8 minutes

Table 2: GooglePlus microbenchmark aggregate plan execution time and added preprocessing time. We vary the optimizer settings with partitioning budgets 1, 8, 64, 512, and 4096 as well as default Postgres optimizer

query in the microbenchmark. Ahead of each such run of five repetitions, we execute an initial untimed cache "warm-up". The resulting aggregate plan execution times may be found in Table 2. While allocating a larger partitioning budget does lead to faster plan execution time, the benefits are not particularly dramatic. In order to better demonstrate the runtime-practicality of our approach over standard cardinality estimation methods, we pivot to the more realistic JOB.

4.2 Plan Execution Time Improvement

We demonstrate that our pessimistic query optimization strategy generates more robust query plans. We investigate two common settings: when the database has precomputed FK indexes, and when it hasn't. We relate this comparison to when the user is experienced versus inexperienced: an experienced user will precompute these indexes dependent on the nature of her workload, while an inexperienced user might not. As in many other systems, Postgres FK indexes are only precomputed through explicit user defined commands and not set as a default. Query plan execution times in the precomputed FK index setting may be found in Figure 9a. Query plan execution times in the FK index absent setting may be found in Figures 9b, and 9c. Each query is run six times in total: the first run to warm up cache and the reported time is the average of the remaining five runs.

In the presence of FK indexes, we find the aggregate plan execution time over all JOB queries to be 3,190 seconds when using the plans generated via Postgres' default cardinality estimates. In contrast, the aggregate execution time for plans generated using bounds is only 1,831 seconds representing a saving of nearly 43%. While most queries yield approximately equivalent plan execution times for those queries that are relatively inexpensive (w.r.t. default Postgres plan execution time), using bounds yields much faster plans for expensive queries. This supports the notion that using bounds leads to more robust query optimization.

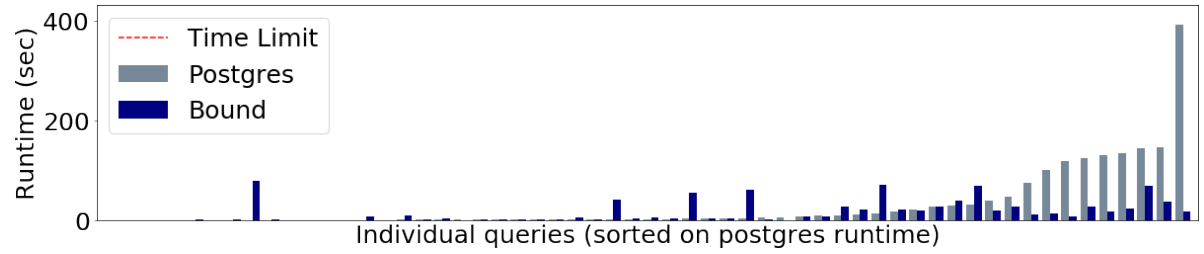
When FK indexes are stripped away, this divide between bound generated and default Postgres generated plans widens. This is because FK accesses have become significantly more costly and more computation must be performed at runtime. We find the aggregate plan execution time over all JOB queries to be 24,725 seconds when using the plans generated via Postgres' default cardinality estimates compared to only 2,304 seconds when using bounds. An order of magnitude difference.

We also place a one hour cutoff on each plan execution time: this limit is enforced on five of the default Postgres plans. Note that only approximately half of all JOB queries are included in Figures 9b, and 9c, for readibility. This is why only three of the five aforementioned time-limit aborted queries appear. While we do not penalize those queries beyond the one hour mark, their actual plan execution time can be significantly longer if not fail entirely. Note the bound generated aggregate plan execution time with and without foreign keys is not significantly different.

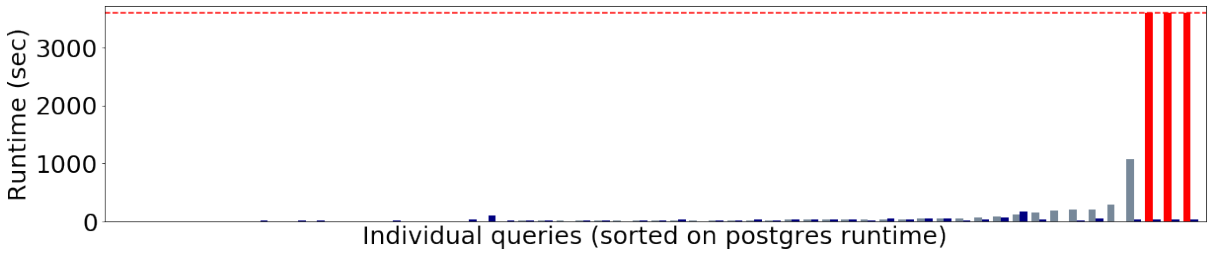
Further investigation into the individual plans suggests that using bounds pushes the optimizer to make more conservative planning decisions. This generally means that the optimizer is more likely to generate shallower, bushy tree plans than the typical left deep. We find that default Postgres severely underestimates intermediate join size higher up the tree. This will naturally encourage the optimizer to generate a left deep plan in order to avoid the cost of building a hash table on, or sorting some intermediate product and instead stream what it predicts will be few intermediate tuples past preexisting base table indexes. This wrong assumption is indeed where default Postgres suffers the most. Alternatively, plans based on bounds suffer the most when the bounds far overshoot and materializing intermediate products and data structures is suboptimal. Nevertheless, we argue that these mistakes are generally less costly at runtime than those associated with underestimation.

We have included a comparison between the RE for our bounds and default Postgres' cardinality estimates in Figure 10a. The PDFs include the relative error of bounds from all the intermediate relations appearing in either the plan produced by using our bounds or from default Postgres. Even though the object of our methods is not to produce highly accurate cardinality estimates, we still enjoy a generally lower RE than default Postgres. In order to compare the tightness of both distributions to the true value, we employ q-error [19].

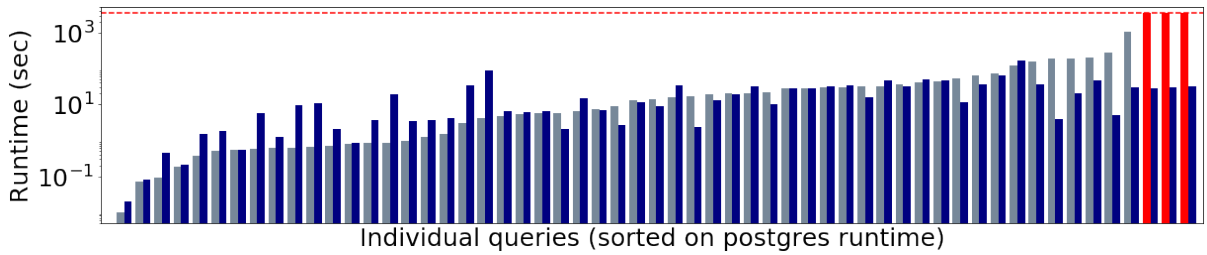
$$\text{q-error}(\text{truth}, \text{estimate}) = \begin{cases} \infty & \text{truth} = 0 \;\&\; \text{estimate} > 0 \\ 1.0 & \text{truth} = 0 \;\&\; \text{estimate} = 0 \\ \max\left(\frac{\text{estimate}}{\text{truth}}, \frac{\text{truth}}{\text{estimate}}\right) & \text{else} \end{cases}$$



(a) Linear scale JOB plan execution time when FK indexes available.



(b) Linear scale JOB plan execution time when FK indexes unavailable.



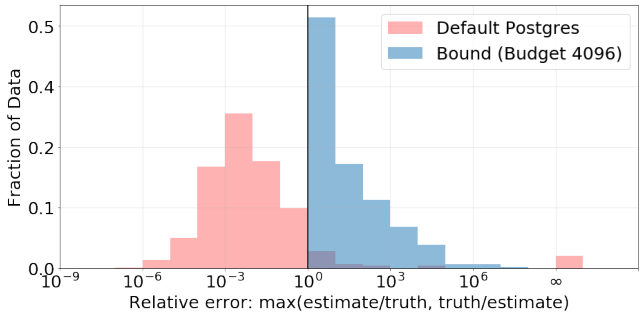
(c) Log scale JOB plan execution time when FK indexes unavailable.

Figure 9: Individual JOB query plan execution times for Postgres optimizer using Postgres' default cardinality estimation (gray) versus optimizers using cardinality bounds (blue). Queries highlighted in red represent those default Postgres plans that failed to complete ahead of the one hour time limit (No plans generated using bounds hit time cutoff). Each graph independently sorts on default Postgres cardinality estimation plan execution time. For readability, only approximately half of all 113 JOB queries are displayed.

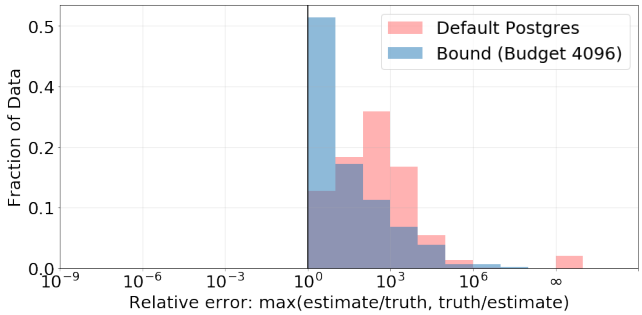
That is, q-error is equivalent to RE when the estimate is greater than the true value. When the estimate is below the true value, we invert the RE. The inverted or equivalent relationship between RE and q-error is highlighted by the similarity in log distributions (see Subfigures 10a, and 10b). Note that when using upper bounds, RE and q-error are equivalent. We include the PDFs, and CDFs of the q-error for our bounds and default Postgres' cardinality estimates in Figures 10b, and 10c, respectively. We highlight that our bounds are consistently tighter to the true value. Furthermore, we emphasize that our bounds are known ahead of time to be single sided (overestimates), whereas default Postgres may produce both overestimates and underestimates which will drive the relative RE between different intermediate relations even further apart. For example, if intermediate relation A is overestimated by a factor of 10, whereas intermediate relation B is underestimated by a factor of 0.1, their relative RE between the two intermediate relations is a factor of 100.

5 RELATED WORK

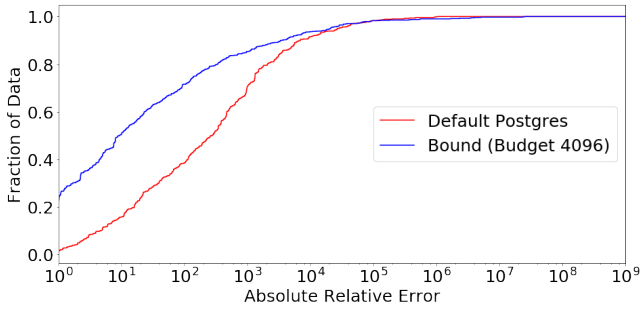
Robust query optimization and bounded query execution [3] is an area focusing not necessarily on picking the fastest plan, but instead places greater value on avoiding poor plans, even at the cost of slightly suboptimal planning and execution time. Babcock et al. propose generating a probability distribution view of a cardinality estimate. The authors allow users to submit a confidence threshold at runtime at which the point query optimizer combines the threshold and cardinality estimate distribution to return a cardinality point estimate with varying aggressiveness [5]. In the context of



(a) PDFs of JOB intermediate product cardinality bounds RE (with partitioning budget 4096) versus default Postgres cardinality estimate RE.



(b) PDFs of JOB intermediate product cardinality bounds q-error (with partitioning budget 4096) versus default Postgres cardinality estimate q-error.



(c) CDFs of JOB intermediate product cardinality bounds q-error (with partitioning budget 4096) versus default Postgres cardinality estimate q-error.

Figure 10: RE and q-error distributions on the JOB.

query re-optimization, Babu et al. propose a confidence interval view of the cardinality estimates, which reduces the need for later query re-optimization [6].

Data sketching is often used to estimate aggregation queries [8]. Since row count is a fundamental aggregation query and is equivalent to cardinality estimation these methods share overlap with our proposed data sketches. However, these techniques are mostly restricted to selectivity estimation on single tables or materialized joins and cannot jump the de-normalization barrier.

Sampling for cardinality estimation is attractive since it inherently handles all common predicate filters while also delivering unbiased estimates. The majority of recent work on cardinality estimation focuses on the generalization and optimization of sampling methods [7, 9, 13, 17, 25]. We describe some of the more recent sampling variations below.

Vanilla uniform Bernoulli sampling has existed for decades in database systems. What is most often offered is a per-table precomputed sample. Let $R(x, y)$ and $T(y, z)$ be two relations with precomputed samples $\mathcal{S}_R(x, y)$ and $\mathcal{S}_T(y, z)$. If an optimizer needs an estimate for a simple join: $R(x, y) \bowtie T(y, z)$, we may use $\mathcal{S}_R$ and $\mathcal{S}_T$ to produce an unbiased estimate:

$$\left|R(x,y) \bowtie T(y,z)\right| \approx \left|\mathcal{S}_R(x,y) \bowtie \mathcal{S}_T(y,z)\right| \cdot \frac{|R|}{|\mathcal{S}_R|} \cdot \frac{|T|}{|\mathcal{S}_T|}$$

This approach can be generalized to queries with arbitrary number of relations and join predicates between them:

$$\left|\bowtie_{j\in[m]} R_j\right| \approx \left|\bowtie_{j\in[m]} \mathcal{S}_{R_j}\right| \cdot \prod_{j\in[m]} \frac{|R_j|}{|\mathcal{S}_{R_j}|}$$

However, in practice this method suffers in the presence of highly selective filters, or many tables appearing in the join. In the case of highly selective filters, the precomputed uniform samples might not have sufficient size (possibly size zero following filter application) and won't be able to capture the true selectivity. In the case of many tables, it is likely that uniform samples won't have sufficient intersection and might lead to empty sample-joins.

Correlated sampling first proposed by Vengerov et al. [25] is a clever twist for generating samples from multiple tables which successfully join. This is the primary drawback of Bernoulli sampling where independent sampling from different tables often yields near empty or empty joins of the samples. Consider the same running example; we wish to estimate the cardinality of $R(x, y) \bowtie T(y, z)$. We fix some sampling probability threshold p and random hash function h mapping the domain of the join attribute to $[0, 1]$ and define sample sets $\mathcal{S}_R$, and $\mathcal{S}_T$ as follows:

$$\mathcal{S}_R = \{r \in R : h(r[y]) < p\}$$
$$\mathcal{S}_T = \{t \in T : h(t[y]) < p\}$$

Because we hold the hash function h constant over the two relations, we have a biased sampling algorithm. We may unbias the sample join as follows:

$$\begin{aligned}\left|R \bowtie T\right| &\approx \left|\mathcal{S}_R \bowtie \mathcal{S}_T\right| \cdot \left(\mathbb{P}\left(r \bowtie t \in \mathcal{S}_R \bowtie \mathcal{S}_T\right)\right)^{-1} \\ &= \left|\mathcal{S}_R \bowtie \mathcal{S}_T\right| \cdot \left(\mathbb{P}(r \in \mathcal{S}_R \wedge t \in \mathcal{S}_T)\right)^{-1} \\ &= \left|\mathcal{S}_R \bowtie \mathcal{S}_T\right| \cdot p^{-1}\end{aligned}$$

Generlizing to multijoin tables: let the join $\bowtie_{j\in[m]} R_j$ feature n join variables. In the case of a simple chain join, we may write $n = m - 1$. In the general case we have:

$$\left| \bowtie_{j\in[m]} R_j \right| \approx \left| \bowtie_{j\in[m]} \mathcal{S}_{R_j} \right| \cdot p^{-n}$$

Two level join sampling is an approach proposed by Chen et al. [7]. The authors categorize the strength and benefits of Bernoulli and Correlated sampling algorithms and seek to combine them with a single "2-level" sampling algorithm. The authors claim that correlated sampling is powerful for capturing correlation between joining columns in separate tables, whereas Bernoulli sampling is useful for capturing correlation between columns within the same table. Correlated and Bernoulli sampling correspond naturally to levels one and two, respectively. Chen et al. describe a per-table single pass algorithm to populate the relation samples.

Index based join sampling, introduced by Leis et. al. [17], seeks to combat the sparse sample problem by replacing the precomputed samples with an iterative process that leverages existing indexes on the full relations to build new samples for intermediate relations at runtime. Specifically, they start with a uniform sample from some base relation in the query and iteratively build outwards along join predicates. In this manner, the method may generate a set of samples for each necessary intermediate result. The algorithm is recursive and ingests some sample $\mathcal{S}$ from an intermediate join or base relation T. The algorithms also ingest a base relation A to be logically joined in to T and which has a precomputed index. The output is a sample $\mathcal{S}'$ from the join $T \bowtie A$. Sample tuples in $\mathcal{S}'$ are generated by using tuples from $\mathcal{S}$ to probe the index on A, returning tuples that will successfully appear in $T \bowtie A$. In order to avoid blowup or starvation inside the intermediate join samples, the authors describe a budgeting and doubling back scheme which is designed to maintain a relatively constant sample size as more tables are logically included.

Improved selectivity estimation [20] combines data sketching (refered to as data synopses in the text), and sampling based methods to improve cardinality estimation. Their work is similar in it's inclusion of data sketching but again is restrictive in their scope. The authors do not extend their methodology to include joins of any kind but instead focus only on multiple filters on a single relation.

Finally, the literature on join tree enumeration is extensive [12, 15, 18, 21, 23]. While the problem of actually enumerating potentially strong physical plans is certainly a valid consideration and can be engineered to pair well with our bounding technique, enumeration does not fall within the scope of our paper.

6 CONCLUSION

Cardinality estimation is a challenging subproblem for cost based query optimizers and remains a primary challenge for even highly sophisticated systems. In particular, in the presence of multijoins. In this work we develop the use of bounds as a practical tool for multijoin query optimization. In doing so, we introduce a method of further tightening state of the art query bounds via random hash partitioning and data sketching. Furthermore, we introduce a novel bound formula restriction scheme that works in tandem with our partition budgeting approach. We demonstrate a tradeoff space between larger partition budget and plan execution time improvement and demonstrate that our bounds can be easily integrated into most cost based systems to achieve robust plans.

This initial work leaves room for future research. For instance, since our work centers on generating more robust query plans, we do not focus on optimization time efficiency. While it should be the possible to generate each necessary statistic with a single pass of the relation in question, our implementation is not optimized for efficiency, and generating the exact degree statistics requires worst case linear additional storage. Instead, it is possible to use approximations. Ting shows promising results in estimating frequent values and their degree with minimal memory impact [24]. Another promising direction is to broaden the scope of applicable queries. In this work, we focus on noncyclic joins but entropic formulas are not restricted in such a way. This would entail broadening Algorithm 2 to reliably generate a larger but still concise subset of all bounding formulas even in the presence of cycles. Finally, using bounds is overkill for some queries. It might instead be possible to use the default optimizer in most cases and then switch to use bounds when a "high-risk" query is detected. This hybrid approach would strike a balance between robust query plans and fast optimization time.

ACKNOWLEDGMENTS

This project is supported by NSF grants AITF 1535565 and III 1614738.

REFERENCES

[1] Amazon. 2018. *IMDb.* https://www.imdb.com/

[2] Austin Appleby. 2008. *SMHasher.* https://github.com/aappleby/smhasher/

[3] Michael Armbrust, Kristal Curtis, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. 2011. PIQL: Success-Tolerant Query Processing in the Cloud. *CoRR* abs/1111.7166 (2011). arXiv:1111.7166 http://arxiv.org/abs/1111.7166

[4] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767. https://doi.org/10.1137/110859440 arXiv:https://doi.org/10.1137/110859440

[5] Brian Babcock and Surajit Chaudhuri. 2005. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. ACM, New York, NY, USA, 119–130. https://doi.org/10.1145/1066157.1066172

[6] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive Re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. ACM, New York, NY, USA, 107–118. https://doi.org/10.1145/1066157.1066171

[7] Yu Chen and Ke Yi. 2017. Two-Level Sampling for Join Size Estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 759–774. https://doi.org/10.1145/3035918.3035921

[8] Graham Cormode. 2010. Sketch Techniques for Approximate Query Processing.

[9] Cristian Estan and Jeffrey F. Naughton. 2006. End-biased Samples for Join Cardinality Estimation. In *Proceedings of the 22Nd International Conference on Data Engineering (ICDE '06)*. IEEE Computer Society, Washington, DC, USA, 20–. https://doi.org/10.1109/ICDE.2006.61

[10] Fedor V. Fomin, Daniel Lokshtanov, Venkatesh Raman, Saket Saurabh, and B.V. Raghavendra Rao. 2012. Faster Algorithms for Finding and Counting Subgraphs. *J. Comput. Syst. Sci.* 78, 3 (May 2012), 698–706. https://doi.org/10.1016/j.jcss.2011.10.001

[11] Google. 2017. *GooglePlus.* https://plus.google.com/

[12] Toshihide Ibaraki and Tiko Kameda. 1984. On the Optimal Nesting Order for Computing N-relational Joins. *ACM Trans. Database Syst.* 9, 3 (Sept. 1984), 482–502. https://doi.org/10.1145/1270.1498

[13] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 631–646. https://doi.org/10.1145/2882903.2882940

[14] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2016. What do Shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? *CoRR* abs/1612.02503 (2016). arXiv:1612.02503 http://arxiv.org/abs/1612.02503

[15] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. 1986. Optimization of Nonrecursive Queries. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 128–137. http://dl.acm.org/citation.cfm?id=645913.671481

[16] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[17] Viktor Leis, Bernharde Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*.

[18] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *In SIGMOD*. 539–552.

[19] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 982–993. https://doi.org/10.14778/1687627.1687738

[20] Magnus Müller, Guido Moerkotte, and Oliver Kolb. 2018. Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses. *Proc. VLDB Endow.* 11, 9 (May 2018), 1016–1028. https://doi.org/10.14778/3213880.3213882

[21] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 677–692. https://doi.org/10.1145/3183713.3183733

[22] Postgres Development Core Team. 2017. *PostgreSQL.* https://www.postgresql.org/

[23] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD '79)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[24] Daniel Ting. 2018. Data Sketches for Disaggregated Subset Sum and Frequent Item Estimation. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 1129–1140. https://doi.org/10.1145/3183713.3183759

[25] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, and Sunil Chakkappen. 2015. Join Size Estimation Subject to Filter Conditions. *PVLDB* 8 (2015), 1530–1541.

A THE AGM BOUND

The proof may be originally found [4] and is reproduced below:

Theorem A.1 (The AGM Bound). *Consider query Q over relational schema σ and let $D = (R_1, \ldots, R_m)$ be a database instance of σ. For all fractional edge covers $(u_1, \ldots, u_m)$ of Q we have*

$$|Q(D)| \leq \prod_{j=1}^{m} |R_j(D)|^{u_j}$$

Proof. Due to the density of rationals in the reals, WLOG we may assume $u_j \in \mathbb{Q}$. Thus there exists $\{v_j\}$ and w s.t. $u_j = v_j/w$ for all j. Let $\sum_j v_j = V$. Consider a possibly repeating collection of subsets $\boldsymbol{a}$, the attributes appearing in Q:

$$\tilde{\boldsymbol{a}}_1, \ldots, \tilde{\boldsymbol{a}}_V \in 2^{\boldsymbol{a}}$$

where the collection contains precisely v_j copies of each subset $\boldsymbol{a}_j$. That is,

$$\left| \left\{ k : \boldsymbol{a}_j = \tilde{\boldsymbol{a}}_k \right\} \right| = v_j$$

We may therefore assume that for all i, the attribute $a_i \in \boldsymbol{a}$ appears in at least w elements of the collection since

$$\left| \{ k : a_i \in \tilde{\boldsymbol{a}}_k \} \right| = \sum_{j : a_i \in \boldsymbol{a}_j} v_j = w \cdot \sum_{j : x \in \boldsymbol{x}_j} u_j \geq w$$

by definition of a fractional edge cover.

Finally, let X be a tuple of random variables

$$\boldsymbol{X} = (X_1, \ldots, X_m)$$

where each X_i corresponds to attribute a_i in $\boldsymbol{a}$. Let X be uniformly distributed on $Q(D)$. That is, for all tuples $t \in Q(D)$, $\mathbb{P}(X = t) = |Q(D)|^{-1}$. Because $\boldsymbol{X}$ is uniformly distributed on a space of size $|Q(D)|$ we have the entropy of $\boldsymbol{X}$

$$h[\boldsymbol{X}] = \log |Q(D)|$$

We may now apply Shearer's Lemma (A.2) on the variable tuple $\boldsymbol{X}$ as well as the collection of attribute subsets $\tilde{\boldsymbol{a}}_k$:

$$b \cdot \log |Q(D)| = b \cdot h[\boldsymbol{X}] \overset{A.2}{\leq} \sum_{k=1}^{A} h\left[\boldsymbol{X}_{\tilde{\boldsymbol{a}}_k}\right] = \sum_{j=1}^{m} v_j h\left[\boldsymbol{X}_{\boldsymbol{a}_j}\right]$$

For each relation R_j, we have the marginal entropy $h[X_{\boldsymbol{a}_j}]$ of the variable tuples on those attributes appearing in R_j is bounded above by the entropy of the uniform distribution on the relational instance $R_j(D)$. We may therefore continue:

$$\sum_{j=1}^{m} v_j h\left[\boldsymbol{X}_{\boldsymbol{a}_j}\right] \leq \sum_{j=1}^{m} v_j \log \left|R_j(D)\right|$$

We conclude:

$$\begin{aligned} |Q(D)| &\leq 2^{\frac{1}{w}\sum_{j=1}^{m} v_j \log |R_j(D)|} \\ &= \prod_{j=1}^{m} \left|R_j(D)\right|^{v_j/w} = \prod_{j=1}^{m} \left|R_j(D)\right|^{u_j} \end{aligned}$$

□

Lemma A.2 (Shearer's Lemma). *Let $\boldsymbol{X} = (X_1, \ldots, X_m)$ be a tuple of random variables and let $\tilde{a}_1, \ldots, \tilde{\boldsymbol{a}}_V$ be a not necessarily distinct collection of subsets of the index set $[m]$ where for each $X_i \in \boldsymbol{X}$, X appears in at least w elements of $\tilde{\boldsymbol{a}}_1, \ldots, \tilde{\boldsymbol{a}}_V$. That is, for all i*

$$|\{k : i \in \tilde{\boldsymbol{a}}_k\}| \geq w$$

For each index subset $I \subset [m]$, define the variable tuple $\boldsymbol{X}_I = (X_i : i \in I)$. We may bound the entropy of X in terms of the marginal entropies of the $X_{\tilde{\boldsymbol{a}}_k}$ and w:

$$w \cdot h[\boldsymbol{X}] \leq \sum_{k=1}^{V} h\left[X_{\tilde{\boldsymbol{a}}_k}\right]$$

B EXAMPLE OF NON-MONOTONICITY OF THE DEGREE BOUND

We wish to demonstrate that the Degree-Bound formula fails to be monotonic non-increasing as the hash size grows. That is, we will construct an explicit example where increasing the hash size will result in a higher join size bound. Note that the bounds presented below differ from the partition budgeting scheme presented in Subsection 3.4. Consider the following conjunctive query:

$$Q(x, y, z) \text{ :- } R(z, y), S(y, z), T(z, w)$$

We populate the relational instances as follows:

x	y
0	0
0	1
1	0
1	1

R

$\ltimes$

y	z
0	0
1	0
2	1
3	1

S

$\ltimes$

z	w
0	0
1	1
2	2
3	3

T

$=$

x	y	z	w
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	0

Q

We begin by considering hash size 1. That is, the generic (non-partitioned) degree bound formula. We have 3 candidate bound formulas:

$$\left|Q(x, y, z)\right| \leq \min \begin{cases} c_R \cdot d_S^y \cdot d_T^z \\ d_R^y \cdot c_S \cdot d_T^z \\ d_R^y \cdot d_S^z \cdot c_T \end{cases}$$

Note the first formula above is in fact tight to the true cardinality of the join:

$$c_{R^{(0)}} \cdot d^y_{S^{(0,0)}} \cdot d^z_{T^{(0)}} = 4 \cdot 1 \cdot 1 = 4$$

We consider a hash size of 2. Define hash function $h(u_i) = i\%2$. That is, simply the modulo-2 function of the attribute value. We define the exact mapping below:

$$h(0) = h(2) = 0$$
$$h(1) = h(3) = 1$$

We may now explicitly describe the hash size 2 degree bound:

$$\left|Q(x, y, z)\right| \leq \sum_{\substack{i,j \\ \in \{0,1\}}} \min \begin{cases} c_{R^{(i)}} \cdot d^y_{S^{(i,j)}} \cdot d^z_{T^{(j)}} \\ d^y_{R^{(i)}} \cdot c_{S^{(i,j)}} \cdot d^z_{T^{(j)}} \\ d^y_{R^{(i)}} \cdot d^z_{S^{(i,j)}} \cdot c_{T^{(j)}} \end{cases} \tag{10}$$

This is where these formulas differ from the budgeting scheme. When using budgeting, the first formula would not introduce partitioning to attribute z and the third formula would not introduce partitioning to attribute y. We have $c_{R^{(i)}} = 2$, $c_{S^{(i,j)}} = 1$, and $c_{T^{(j)}} = 2$ for all values $i, j \in \{0, 1\}$. This implies that Equation 10 will be calculated as follows:

$$\begin{aligned} &\sum_{\substack{i,j \\ \in \{0,1\}}} \min \begin{cases} c_{R^{(i)}} \cdot d^y_{S^{(i,j)}} \cdot d^z_{T^{(j)}} \\ d^y_{R^{(i)}} \cdot c_{S^{(i,j)}} \cdot d^z_{T^{(j)}} \\ d^y_{R^{(i)}} \cdot d^z_{S^{(i,j)}} \cdot c_{T^{(j)}} \end{cases} \\ \leq &\sum_{\substack{i,j \\ \in \{0,1\}}} \min \begin{cases} 2 \cdot 1 \cdot 1 \\ 2 \cdot 1 \cdot 1 \\ 2 \cdot 2 \cdot 2 \end{cases} = \sum_{\substack{i,j \\ \in \{0,1\}}} 2 = 8 \end{aligned}$$

Observe that the bound has increased despite the fact that hash size has increased. Moreover, since we have gone from hash size 1 to hash size 2, each of the partitions in the hash size 2 sketches are strict sub-partitions of the partitions in

the hash size 1 sketches. This is in contrast to a non strict sub-partitioning (i.e. where tuples are mixed between buckets during a change in hash size).

C SINGLE PASS ALGORITHM TO COMPUTE BOUND SKETCH

The downside of using the BS is the increased optimization time. We populate our sketches using a naive algorithm based on the SQL query found in Figure 11 which we feed to our modified postgres instance. While this method is sufficient to demonstrate that more robust plans are possible, it is not optimized for efficient optimization time. In the presence of FK indexes, the additional optimization time over the JOB is 4,795 seconds. This additional optimization time is longer than execution time. Without FK indexes, the additional optimization time is 6,450 seconds. Again, the additional optimization time is longer than actual execution time but insignificant compared to the execution time for plans generated by default Postgres.

The BS may calculated in a single pass following the Algorithm 3. For simplicity, we assume a binary relation R with two join variables x, y and hash partition sizes M_x, M_y. However, this method is easily generalized to any number of join variables and may be executed concurrently in the same table scan but with alternative hash partition sizes.

Algorithm 3 Bound Sketch Generator.

procedure BOUND SKETCH($R(x,y), M_x, M_y$) ▷ input relation $R(x,y)$ with hash partitions sizes M_x, M_y.
  $c_R \leftarrow [0]_{M_x \times M_y}$ ▷ the count tensor, and both degree tensors are initiated as $M_x \times M_y$ tensors of zeros.
  $d_R^x \leftarrow [0]_{M_x \times M_y}$
  $d_R^y \leftarrow [0]_{M_x \times M_y}$
  $\text{HM}_x \leftarrow$ new hashmap: $([M_y], W) \to \mathbb{Z}_+$
  $\text{HM}_y \leftarrow$ new hashmap: $([M_x], W) \to \mathbb{Z}_+$
  for $t \in R(x,y)$ **do**
    $m_x = H(t[x])$
    $m_y = H(t[y])$
    $c_R[m_x, m_y] + +$
    $\text{HM}_x(m_y, t[x]) + +$
    $\text{HM}_y(m_x, t[y]) + +$
    if $\text{HM}_x(m_y, t[x]) > d_R^x[m_x, m_y]$ **then**
      $d_R^x[m_x, m_y] \leftarrow \text{HM}_x(m_y, t[x])$
    if $\text{HM}_y(m_x, t[y]) > d_R^y[m_x, m_y]$ **then**
      $d_R^y[m_x, m_y] \leftarrow \text{HM}_y(m_x, t[y])$
  return (c_R, d_R^x, d_R^y) ▷ return sketches

While this algorithm is linear in runtime, it also requires linear additional storage in the worst case. Alternatively, one degree sketch (with respect to x) as well as the count sketch could also be populated using the nested SQL query found in Figure 11. To populate both degree sketches, a query of this form would need to be executed twice. However, it is often the case that only a single degree statistic is needed for all bounding formulas in which case the degree sketch with respect to y need not be calculated.

```
SELECT
    R_inner.hx AS hx
    R_inner.hy AS hy,
    SUM(R_inner.cnt) AS cnt,
    MAX(R_inner.cnt) AS max_degree
FROM(
    SELECT
        hash(x) AS hx,
        hash(y) AS hy,
        x,
        COUNT(*) AS cnt
    FROM R
    GROUP BY hx, hy, x) AS R_inner
GROUP BY hx, hy;
```

Figure 11: Nested SQL query used to generate BS.

The primary downside of using the BS is the increased optimization time. We populate our sketches using a naive algorithm based on the SQL query found in Figure 11 which we feed to our modified postgres instance. In the case of multiple join attributes in a single table, we must submit the query multiple times. While this method is sufficient to demonstrate that more robust plans are possible, it is not optimized for efficient optimization time. In the presence of FK indexes, the additional optimization time over the JOB is 4,795 seconds. This additional optimization time is longer than the plan execution time for both default Postgres and using our bounds. Without FK indexes, the additional optimization time is 6,450 seconds. Again, the additional optimization time is longer than plan execution time for the plans generated by bounds, but insignificant compared to the plan execution time for plans generated by default Postgres.

D GOOGLEPLUS MICROBENCHMARK TEMPLATE EXAMPLES

```
SELECT COUNT(*)
FROM
    community_44 AS t0,
    community_44 AS t1,
    community_44 AS t2,
    community_44 AS t3
WHERE
    t0.object = t1.subject AND
    t1.object = t2.subject AND
    t2.object = t3.subject AND
    t0.subject % 512 = 89 AND
    t3.object % 512 = 174;
```

Figure 12: Template 4a, Googleplus Community 44

```
SELECT COUNT(*)
FROM
    community_30 AS t0,
    community_30 AS t1,
    community_30 AS t2,
    community_30 AS t3,
    community_30 AS t4
WHERE
    t0.object = t1.subject AND
    t0.object = t2.subject AND
    t0.object = t3.subject AND
    t3.object = t4.subject AND
    t0.subject % 256 = 49 AND
    t1.object % 256 = 213 AND
    t2.object % 256 = 152 AND
    t4.object % 256 = 248;
    AND ci.movie_id = mc.movie_id;
```

Figure 13: Template 5c, Googleplus Community 30

```
SELECT COUNT(*)
FROM
    community_5 AS t0,
    community_5 AS t1,
    community_5 AS t2,
    community_5 AS t3,
    community_5 AS t4
WHERE
    t0.object = t1.subject AND
    t1.object = t2.subject AND
    t2.object = t3.subject AND
    t2.object = t4.subject AND
    t0.subject % 1024 = 615 AND
    t3.object % 1024 = 765 AND
    t4.object % 1024 = 384;
```

Figure 14: Template 5e, Googleplus Community 5