

THE NATIONAL UNIVERSITY
of SINGAPORE



School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TR12/10

***"UpSizeR: Synthetically Scaling an Empirical
Relational Database"***

***Y. C. Tay, Bingtian Dai, Tao Wang, Yang Sun,
Yong Lin and Yuting Lin***

December 2010

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

UpSizeR: Synthetically Scaling an Empirical Relational Database*

Y.C. Tay
National University of
Singapore
dcstayyc@nus.edu.sg

Eldora Y. Sun
National University of
Singapore
sunyang@comp.nus.edu.sg

Bing Tian Dai
Singapore Management
University
daibingtian@gmail.com

Yong Lin
National University of
Singapore
linyong@nus.edu.sg

Daniel T. Wang
Teradata
Daniel.Wang@teradata.com

Yuting Lin
National University of
Singapore
linyuting@comp.nus.edu.sg

ABSTRACT

This paper presents UpSizeR, a software that takes as input an empirical relational dataset $\mathcal{D}$ and a scale factor s , and generates a synthetic dataset $\tilde{\mathcal{D}}$ that is similar to $\mathcal{D}$ but s times its size. Such a tool can be useful for scaling up $\mathcal{D}$ for scalability testing ($s > 1$), scaling down for application debugging ($s < 1$), or anonymization ($s = 1$).

Experiments with Flickr show that query results and response times on UpSizeR output match those on crawled data. They also accurately predict throughput degradation for a scale out test.

1. INTRODUCTION

A database managements system for an enterprise is a complicated collection of software and hardware. Its complexity and its critical role require that a different index design, a scale out of machines, a new business application, etc., be adequately tested before deployment. Such testing needs to use a dataset of an appropriate size.

One possibility is to use a TPC¹ benchmark for such tests. TPC datasets can be scaled to desired sizes, and are also domain-specific: TPC-C for online transaction processing, TPC-H for decision support, etc. Vendors have used these benchmarks to improve and compare their products, and researchers have used them to test and compare their algorithms and prototypes. The TPC benchmarks have thus played an important role in the growth of the database industry and the progress of database research.

However, while there are myriad database applications, there are only a few TPC benchmarks. A TPC benchmark

is not equally relevant to two different applications within its domain and, at any moment, there are many applications not covered by the benchmarks. This situation can only get worse, as the proliferation of new database applications far outpaces the approval of new TPC benchmarks.

1.1 Problem Statement

There is thus a pressing need for a tool to help database owners generate application-specific datasets to specified size. We state this issue as the **Dataset Scaling Problem**:

Given a set of relational tables $\mathcal{D}$ and a scale factor s , generate a database state $\tilde{\mathcal{D}}$ that is similar to $\mathcal{D}$ but s times its size.

This paper presents **UpSizeR**, a first-cut tool for solving the above problem.

One can define “ s times its size” in various ways (number of tuples or bytes, etc.), and numerical precision is unnecessary — if $s = 3$, it would not matter if the generated $\tilde{\mathcal{D}}$ were actually 3.14 times $\mathcal{D}$ ’s size (however defined).

Rather, the issue here is “similarity”: $\tilde{\mathcal{D}}$ must reflect relationships among the columns and rows of $\mathcal{D}$. Instead of measuring similarity in the data itself (with some statistical test or graph property, say), we assume the database owner has some set of queries $\mathcal{Q}$ on hand, and she will judge if $\tilde{\mathcal{D}}$ is similar to $\mathcal{D}$ by running $\mathcal{Q}$ on both and comparing the results (number of tuples generated, aggregated values computed, query response time, etc.).

1.2 Motivation for $s > 1$, $s = 1$, $s < 1$

There are various possibilities for why one might want to synthetically scale up ($s > 1$) an empirical dataset. Some web applications have user populations that grow at break-neck speed (one recent example being Animoto²), so a small but fast-growing service may need to test the scalability of their hardware and software architecture with larger versions of their datasets.

Another example is where an enterprise supplies a vendor with only a sample of its dataset (e.g. the entire dataset is too large for easy transfer), and the vendor needs to scale up the sample to an appropriate size.

*This research was supported in part by Ministry of Education grant MOE2008-T2-1-095.

¹<http://www.tpc.org/>

²<http://animoto.com>

Taking a small sample of a large dataset is itself nontrivial. For example, if a dataset contains 2000000 buyers, and we want to extract a sample with 1000 buyers, it does not suffice to randomly pick 1000 buyers; e.g. we may need to add their suppliers’ other buyers, and this recursive adding can grow the sample to an indeterminate size. Instead, one can use UpSizeR with $s < 1$ to downsize the dataset.

An enterprise may want to downsize its dataset, not just for a vendor, but for itself. For example, rather than debug a new application by running it on a production dataset, one can use UpSizeR to get a small synthetic copy for testing.

In providing a vendor with just a small sample of its dataset, an enterprise may be motivated by privacy or proprietary considerations. UpSizeR also addresses such issues, since its output dataset is synthetic. Thus, for $s = 1$, UpSizeR can be viewed as making an anonymized copy of a dataset. Note, however, that some information leakage is inevitable since $\tilde{\mathcal{D}}$ is, after all, similar to $\mathcal{D}$.

Such anonymization can be useful for, say, exploring different system configurations or implementations in the cloud (leveraging on its elasticity, and before investing in a particular configuration). Instead of exposing their real dataset (i.e. their crown jewels), an enterprise can use UpSizeR to upload a synthetic copy into the cloud.

1.3 Related Work

Scalability engineering requires an exploration of the design space for load balancing, query plans, power budgeting, etc. For example, the designers of SCOPE (a SQL-like scripting language for parallel processing of massive data [7]) exercised it with scalability tests that use TPC-H. However, Seltzer et al. [17] have observed how standard benchmarks can be irrelevant for particular applications, and argued for application-specific benchmarking. For database systems, this approach must start with application-specific datasets.

The TPC benchmarks have specific scaling rules, and a table is generated by independently generating the required number of tuples. The decision to generate completely synthetic relations, instead of modeling empirical relationships, can be traced back to the Wisconsin benchmark [9]. Bitton et al. made that choice then because (i) very large amounts of data are needed for the empirical values to reflect their underlying distribution; (ii) synthetic data facilitates query design (for desired selectivity factors, join sizes, etc.); and (iii) empirical datasets are hard to scale.

Nowadays, (i) is not an issue, since most datasets are big. In fact, gargantuan datasets are so common that the case for scale factor $s < 1$ may be more important than $s > 1$. The task for UpSizeR is to scale a dataset; we assume the dataset owner already has an application-specific query set $\mathcal{Q}$, so we do not have to design the queries and (ii) is not a problem. Issue (iii) remains true, i.e. the Dataset Scaling Problem is hard, but twenty-seven years have passed and it is time to revisit the problem.

To see why this task is nontrivial, consider the toy $\mathcal{D}$ in Fig. 1. An obvious possibility for $s = 2$ is to scale $\mathcal{D}$ to $\tilde{\mathcal{D}}$ by making a copy. However, scaling this way may violate attribute constraints (case (a) in Fig. 1) or join sizes (case (b)). Besides, copying does not work for $s < 1$.

For $s = 1$, related work on anonymization (k -anonymity, etc.) mostly focuses on a single table, or allow at most one foreign key per table [16]. There is recent work on anonymization that preserves semantic constraints [20] or

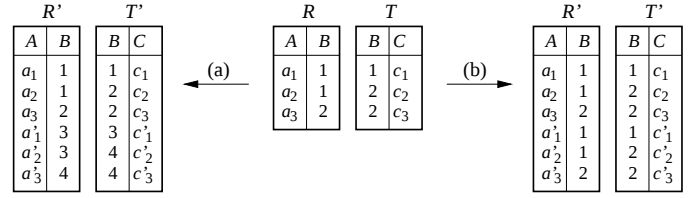


Figure 1: $\mathcal{D} = \{R, T\}$, $\mathcal{D}' = \{R', T'\}$, $s = 2$. Naive copying does not work: For (a), creating new values may violate constraints on B (e.g. value range or number of distinct values); for (b), without creating new values, the scale up in join sizes may be wrong.

query plans [6]; UpSizeR can adopt some of their techniques to address those issues for $s \neq 1$.

The TPC approach is being adopted by a new generation of benchmarks [2, 8]. So far, the use of empirical data is very limited. For example, MUDD only extracts names and addresses from a real dataset [18], while TEXTURE [11] extracts word distribution, document lengths, etc. from “seed” documents and use them to independently generate synthetic documents (like how TPC generates tuples).

Although the UpSizeR user may have a query set $\mathcal{Q}$ on hand, the UpSizeR version presented here does not use $\mathcal{Q}$. Binnig et al.’s reverse query processing [3] uses query results to generate a *smallest* dataset to test the application, whereas QAGen uses a given query plan with size constraints to generate a corresponding dataset [4], without requiring similarity to real data. No doubt, UpSizeR can be improved through analyzing the application’s queries, and we will discuss this in Sec. 7.

1.4 Our Contribution

This paper makes three contributions to the Dataset Scaling Problem:

1. Sec. 3 presents UpSizeR’s algorithms for scaling an empirical dataset.
2. We released our UpSizeR implementation for open-source development³. Our wish is that UpSizeR can work for any relational database. However, given the diversity of applications, the complexities in real data and the pressing need for a scaling tool, UpSizeR development into an industrial-strength tool requires community effort.
3. Sec. 5 identifies an Attribute Correlation Problem for social networks. This problem (not addressed in this paper) calls for database-theoretic studies of how social interactions affect inter-column and inter-row correlations in relational databases. We believe this is a rich, new area for database research.

1.5 Paper Overview

We begin in Sec. 2 by introducing our notation and stating our assumptions. Sec. 3 then presents the UpSizeR algorithms. Sec. 4 describes how the assumptions can be relaxed, while Sec. 5 points out some limitations. Sec. 6 validates UpSizeR by comparing it to real data, and demonstrates how it can predict throughput degradation in a scale out test.

³<http://www.comp.nus.edu.sg/~upsizer>

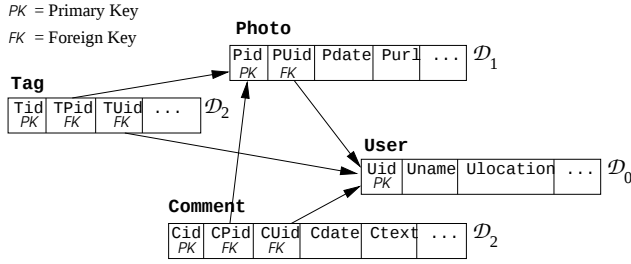


Figure 2: A small schema graph for a photograph database $\mathcal{F}$. Photo records the owners (Puid) who uploaded the photographs, Comment records the comments on photographs (CPid) and their authors (Cuid), and Tag records the tags on photographs (TPid) and the users who specified the tags (Tuid). User records these owners, authors and taggers. Sorting this graph into $\mathcal{D}_i$ (Sec. 3) gives $\mathcal{D}_0 = \{\text{User}\}$, $\mathcal{D}_1 = \{\text{Photo}\}$ and $\mathcal{D}_2 = \{\text{Comment}, \text{Tag}\}$. $\mathcal{D}_1$ has one equivalence class $[\{\text{Uid}\}] = \{\text{Photo}\}$ and $\mathcal{D}_2$ also has one equivalence class $[\{\text{Pid}, \text{Uid}\}] = \{\text{Comment}, \text{Tag}\}$ (see [K] definition in Sec. 3).

Sec. 7 reviews more related work, before Sec. 8 concludes with a summary and a description of future work.

2. UPSIZER SPECIFICATION

We first fix our terminology and notation in Sec. 2.1 and list our assumptions in Sec. 2.2. We then describe the input and output for UpSizeR in Sec. 2.3.

2.1 Terminology and Notation

We assume the reader is already familiar with the relevant definitions, and the following only serves to state our choice of terminology and notation.

A **database state** $\mathcal{D}$ consists of a set of **tables**. Each table has a **relation scheme**, a corresponding relation, and a **primary key**. The relation scheme is a set of **attributes**, including the primary key. The relation is a set of **tuples**, each of which assigns **values** to the attributes.

If a primary key K of table T appears as an attribute K' in another table T' , K' is a **foreign key**. Such a K defines an **edge** from T' to T . These edges form a directed **schema graph** for the database state $\mathcal{D}$.

Fig. 2 gives an example of a schema graph for a database $\mathcal{F}$, like Flickr⁴, that stores photographs uploaded by, commented upon and tagged by a community of users.

Each edge in the schema graph induces a bipartite graph between T and T' , with bipartite edges between a tuple in T with K value v and the tuples in T' with K' value v . The number of such edges is denoted $\deg(v, T')$. This is illustrated in Fig. 3 for $\mathcal{F}$.

For a positive number s , to **scale up** $\mathcal{D}$ by s is to generate a synthetic database state $\tilde{\mathcal{D}}$ such that:

- (S1) $\tilde{\mathcal{D}}$ has the same schema as $\mathcal{D}$.
- (S2) $\tilde{\mathcal{D}}$ and $\mathcal{D}$ are similar in terms of query results.
- (S3) For each table T_0 that has no foreign key, the number of T_0 tuples in $\tilde{\mathcal{D}}$ should be s times that in $\mathcal{D}$; the sizes of tables with foreign keys are indirectly determined through their foreign key constraints.

⁴<http://www.flickr.com>

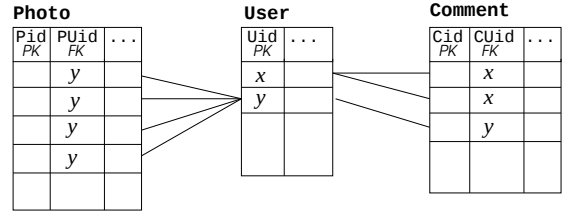


Figure 3: A schema graph edge in Fig. 2 from Photo to User for the key Uid induces a bipartite graph between the tuples of User and Photo. Here, $\deg(x, \text{Photo}) = 0$ and $\deg(y, \text{Photo}) = 4$. Similarly, $\deg(x, \text{Comment}) = 2$ and $\deg(y, \text{Comment}) = 1$.

How should one measure similarity of $\tilde{\mathcal{D}}$ and $\mathcal{D}$? What counts as similar for one application may not be so for another. Since one motivation for UpSizeR lies in its use for scalability studies, UpSizeR should provide accurate forecasts of storage requirement, query time and retrieval results for larger datasets. The latter two are possible similarity measures, and they require some set $\mathcal{Q}$ of test queries.

We hence assume that the UpSizeR user has such a $\mathcal{Q}$ on hand (in addition to the database state $\mathcal{D}$) to measure the similarity of $\tilde{\mathcal{D}}$ and $\mathcal{D}$, in terms of tuples retrieved, aggregates computed, response time, etc. This definition of dataset similarity, in terms of some user-specified $\mathcal{Q}$ and result similarity, makes (S2) application-specific.

For (S3), if $\mathcal{D}$ has the schema in Fig. 2, where **User** has no foreign keys, then the number of **User** tuples in $\tilde{\mathcal{D}}$ should be s times the number of **User** tuples in $\mathcal{D}$. The number of **Photo** tuples in $\tilde{\mathcal{D}}$ will be determined by $\deg(\text{Uid}, \text{Photo})$, and the size of **Comment** then determined by the correlated values of $\deg(\text{Pid}, \text{Comment})$ and $\deg(\text{Uid}, \text{Comment})$.

One central issue in scaling up $\mathcal{D}$ lies in replicating its empirical inter-key correlations. For example, in the **Comment** table of $\mathcal{F}$, an author is more likely to comment on her own photographs. This implies that $\deg(\text{Uid}, \text{Comment})$ and $\deg(\text{Uid}, \text{Photo})$ are correlated. For each primary key K , let f_K be the joint degree distribution, i.e.

$$f_K(d_1, \dots, d_r) = \Pr(\deg(v, T_1) = d_1, \dots, \deg(v, T_r) = d_r),$$

where the random variable is K value v , and $T_1, \dots, T_r$ are the tables that contain K as foreign key.

Furthermore, if users are clustered into chefs, writers, etc., and photographs are clustered into cars, cakes, etc., then chefs are more likely to comment on photographs of cakes.

For a table T with two foreign keys, let X be the random variable for a foreign key value, where these values are divided into clusters $c_1^X, \dots, c_m^X$. Let random variable Y and clusters $c_1^Y, \dots, c_n^Y$ be similarly defined for the other foreign key. Each $\langle c_i^X, c_j^Y \rangle$ is called a **co-cluster** [10], with joint distribution f_T^{cc} induced by T , i.e.

$$f_T^{cc}(c_i^X, c_j^Y) = \Pr(X \in c_i^X, Y \in c_j^Y).$$

Thus, in $\mathcal{F}$, **Comment** induces a co-clustering for photographs and users through a joint distribution f_{Comment}^{cc} .

Finally, we refer to generation of values for non-key attributes as **content generation**.

We will use v , T and $\deg(v, T')$ to denote a value, table or degree in the given $\mathcal{D}$, and $\tilde{v}$, $\tilde{T}$ and $\deg(\tilde{v}, \tilde{T}')$ to denote their synthetically generated counterparts in $\tilde{\mathcal{D}}$.

2.2 Assumptions

To start, we assume the following:

- (A1) Each primary key is a singleton attribute.
- (A2) A table has at most two foreign keys.
- (A3) The schema graph is acyclic.
- (A4) The degree distribution is static. (E.g. the number of comments per user has the same distribution in $\mathcal{F}$ and $\tilde{\mathcal{F}}$.)
- (A5) Non-key attribute values for a tuple t only depends on t 's key values.
- (A6) Key values only depend on the joint degree and co-clustering distributions. (This is not true for $\mathcal{F}$ — see Sec. 5.)

These assumptions are mostly adopted as a tradeoff, giving up technical generality for expository clarity. In Sec. 4, we discuss how they can be relaxed.

2.3 Input and Output

The input to UpSizeR is given by an empirical dataset $\mathcal{D}$ and a positive number s that specifies the **scale factor**.

In response, UpSizeR will generate a syntactic database state $\tilde{\mathcal{D}}$ satisfying (S1), (S2) and (S3) — see Sec. 2.1. The ratio in size of $\tilde{\mathcal{D}}$ to $\mathcal{D}$ is only approximately s , since the exact size of $\tilde{\mathcal{D}}$ is determined by key constraints, schema semantics (e.g. certain tables may have fixed sizes) and randomness in tuple generation.

The main issue in the Dataset Scaling Problem is similarity. For UpSizeR to be generally applicable, similarity must be application-specific. By defining dataset similarity in terms of query results (instead of, say, statistical distributions or graph properties), (S2) gives the UpSizeR user the final say.

3. UPSIZER ALGORITHMS

We now describe the UpSizeR algorithms, using $\mathcal{F}$ as an example. The Appendix provides more details in pseudocode.

3.1 Extract probability distributions

For each table T in $\mathcal{D}$, UpSizeR processes T to extract the joint degree distribution f_K , where K is the primary key of T (see Sec. 2.1). This is done by normalizing the frequency distribution obtained from T . From f_K , UpSizeR can derive various marginal distributions, as needed, including that for $\deg(v, T')$ where v is a K value and T' is any table with K as foreign key.

If T has more than one foreign key, UpSizeR then extracts the co-clustering distribution f_T^{cc} among the foreign keys. UpSizeR can work with any co-clustering algorithm (for the experiments, we use the one by Dhillon et al. [10]).

3.2 Sort the tables

Recall from (A3) that we assume the schema graph is acyclic. UpSizeR first groups the tables in $\mathcal{D}$ into subsets $\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2, \dots$ by sorting this graph, in the following sense:

- all tables in $\mathcal{D}_0$ have no foreign keys;
- for $i \geq 1$, $\mathcal{D}_i$ contains tables whose foreign keys are primary keys in $\mathcal{D}_0 \cup \mathcal{D}_1 \cup \dots \cup \mathcal{D}_{i-1}$.

For $\mathcal{F}$, $\mathcal{D}_0 = \{\text{User}\}$, $\mathcal{D}_1 = \{\text{Photo}\}$ and $\mathcal{D}_2 = \{\text{Comment}, \text{Tag}\}$; here, the tables in $\mathcal{D}_i$ coincidentally have i foreign keys. This is not true in general. For the TPC-H example in the Appendix, the tables in $\mathcal{D}_2$ have just 1 foreign key each.

3.3 Partition $\mathcal{D}_i$ into equivalence classes $[\mathbf{K}]$

For each $\mathcal{D}_i$ and set of foreign keys $\mathbf{K}$, let $[\mathbf{K}]$ be the set of all tables in $\mathcal{D}_i$ with $\mathbf{K}$ as foreign keys. $[\mathbf{K}]$ is thus an equivalence class, and each $\mathcal{D}_i$ is partitioned by $[\mathbf{K}]$.

In the $\mathcal{F}$ example, $\mathcal{D}_1$ has one equivalence class $[\{\text{Uid}\}] = \{\text{Photo}\}$, and $\mathcal{D}_2$ also has one equivalence class $[\{\text{Pid}, \text{Uid}\}] = \{\text{Comment}, \text{Tag}\}$. In general, $\mathcal{D}_i$ may have more than 1 equivalence class. (For the TPC-H example in the Appendix, $\mathcal{D}_3$ has 2 equivalence classes.)

We need this definition because the tables in $[\mathbf{K}]$ are correlated through $\mathbf{K}$, so they are generated together.

3.4 Generate T in $\mathcal{D}_0$

Suppose T in $\mathcal{D}_0$ has h tuples. Since T has no foreign keys, UpSizeR simply generates sh primary key values for $\tilde{T}$. For example, the **User** table in $\tilde{\mathcal{F}}$ has s times the number of **Uids** in $\mathcal{F}$.

Recall assumption (A5) that non-key values of a tuple depend only on its key values. For $\mathcal{D}_0$, this means the non-key attributes can be independently generated (without regard to the primary key values, which are arbitrary) by some content generator.

For example, values for **Uname** and **Ulocation** in $\tilde{\mathcal{F}}$ can be picked from sets of names and locations, according to frequency distributions extracted from $\mathcal{F}$.

3.5 UpSizeR's main loop

A loop now generates tables in $\mathcal{D}_1, \mathcal{D}_2, \dots$ in that order. The loop terminates when all tables are generated.

Each loop first generates $\deg(\tilde{v}, \tilde{T}')$ before generating $\tilde{T}'$.

3.5.1 Generate $\deg(\tilde{v}, \tilde{T}')$

In our $\mathcal{F}$ example, $\deg(\tilde{u}, \text{Photo})$ and $\deg(\tilde{u}, \text{Comment})$ are correlated, since a user $\tilde{u}$ is likely to comment on her own photographs. With the $\mathcal{D}_i$ ordering, **Photo** is scaled up before **Comment**. To generate $\deg(\tilde{u}, \text{Comment})$, we must therefore use f_{Uid} to determine

$$Pr(\deg(u, \text{Comment}) = d' \mid \deg(u, \text{Photo}) = d).$$

In general, suppose $\tilde{T}'$ has the set of foreign keys $\mathbf{K}$, $K \in \mathbf{K}$ and K is the primary key of T . By the ordering $\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2, \dots$, $\tilde{T}'$ would already have been generated, so the synthetic K values $\tilde{v}$ are ready to be used as foreign key values to generate $\tilde{T}'$.

Let $T'_1, \dots, T'_n$ be the tables that have $\mathbf{K}$ as foreign keys, i.e. $[\mathbf{K}] = \{T'_1, \dots, T'_n\}$. Earlier iterations of UpSizeR's main loop may have generated $\deg(\tilde{x}, \tilde{T}''_1), \dots, \deg(\tilde{x}, \tilde{T}''_m)$ for all K values $\tilde{x}$ and some tables $\tilde{T}''_1, \dots, \tilde{T}''_m$. UpSizeR now derives from the joint degree distribution f_K the conditional

$$Pr(\deg(\tilde{v}, T'_1) = d'_1, \dots, \deg(\tilde{v}, T'_n) = d'_n \mid \deg(\tilde{v}, \tilde{T}''_1) = d_1, \dots, \deg(\tilde{v}, \tilde{T}''_m) = d_m).$$

This distribution is then used to choose $\deg(\tilde{v}, \tilde{T}'_i)$.

3.5.2 Generate $\tilde{T}'$ in $\mathcal{D}_i$ for $i \geq 1$

By (A2), each T' in $\mathcal{D}_i$ ($i \geq 1$) has 1 or 2 foreign keys.

Case T' has 1 foreign key:

Suppose T' has foreign key set $\mathbf{K} = \{K\}$, where K is primary key of T . In the $\mathcal{F}$ example, **Photo** has $\mathbf{K} = \{\text{Uid}\}$ and **User** is generated first; for each **Uid** $\tilde{v}$, we then generate $\deg(\tilde{v}, \text{Photo})$ tuples for **Photo**.

In general, for each $\tilde{v}$, we generate $\deg(\tilde{v}, \tilde{T}')$ tuples of $\tilde{T}'$, using $\tilde{v}$ for their K value and arbitrary (but unique) values for their primary key. Each tuple's non-key values are then assigned by content generation.

Case T' has 2 foreign keys:

Suppose T' has foreign key set $\mathbf{K} = \{K_1, K_2\}$ and K_i is the primary key of T_i for $i = 1, 2$. For $\mathcal{F}$, **Comment** has $\mathbf{K} = \{\text{Pid}, \text{UId}\}$, so we need to use the distribution f_{Comment}^{cc} that co-clusters **Pids** and **Uids**.

For $i = 1, 2$, UpSizeR first assigns every previously generated K_i value $\tilde{v}_i$ to a cluster cc_i , then assigns $\deg(\tilde{v}_i, \tilde{T}')$ according to the degree distribution. We then force $\sum_{\tilde{v}_1} \deg(\tilde{v}_1, \tilde{T}') = \sum_{\tilde{v}_2} \deg(\tilde{v}_2, \tilde{T}')$ by randomly incrementing nonzero degrees in the smaller sum.

To generate a new tuple t for $\tilde{T}'$, UpSizeR generates a new primary key value $\tilde{v}'$ for $\tilde{T}'$ and assigns $\tilde{v}'$ to a random $\langle cc_1, cc_2 \rangle$ according to $f_{T'}^{cc}$. Within this co-cluster, UpSizeR picks $\tilde{v}_i$ in cc_i with probability proportional to $\deg(\tilde{v}_i, \tilde{T}')$.

The key values $\tilde{v}, \tilde{v}_1, \tilde{v}_2$ now suffice to generate the rest of t . $\deg(\tilde{v}_1, \tilde{T}')$ and $\deg(\tilde{v}_2, \tilde{T}')$ are then decremented before generating the next tuple. This loop terminates when all $\deg(\tilde{v}_i, \tilde{T}')$ are 0.

4. RELAXING THE ASSUMPTIONS

We adopted the strong assumptions above to help simplify the presentation. We now explain how (A1)–(A5) can be relaxed.

4.1 Composite primary keys

We can relax the assumption (A1) that primary keys are not composite by adding new attributes. If a table T has, say, attributes B_1 and B_2 that together act as the primary key, we add a new attribute C to take over as primary key. This is what we do for the TPC-H example in the Appendix.

To do so, we must ensure that $\langle B_1, B_2 \rangle$ values are unique in $\tilde{T}'$. If B_1 and B_2 are foreign keys and there are degree constraints $\deg(\tilde{v}_1, \tilde{T}')$ and $\deg(\tilde{v}_2, \tilde{T}')$ from B_1 and B_2 values $\tilde{v}_1$ and $\tilde{v}_2$ that are generated first, then we have to check for repetitions and resample when they occur.

4.2 More than two foreign keys

Assumption (A2) states that a table should have at most two foreign keys. UpSizeR can generate a table $\tilde{T}'$ with d foreign keys for $d \geq 3$, as long as there is an algorithm for co-clustering in d dimensions,

For example, if $d = 3$, we use any 3-dimensional co-clustering algorithm to generate the clusters, then use the degree distributions to assign $\deg(\tilde{v}, \tilde{T}')$. A new tuple is then assigned a co-cluster $\langle cc_1, cc_2, cc_3 \rangle$, using the co-clustering distribution $f_{T'}^{cc}(\langle cc_1, cc_2, cc_3 \rangle)$. The foreign key values $\tilde{v}_i$ in cc_i are then chosen according to $\deg(\tilde{v}_i, \tilde{T}')$.

4.3 Cyclic schemas

UpSizeR uses the acyclicity assumption (A3) to sort the tables into $\mathcal{D}_0, \mathcal{D}_1, \dots$ and generate them in that order. The simplest violation of (A3) is where there is a self-loop. The most important example of this is where a table T has employee number **Eid** as primary key and manager **Mid** as foreign key (since every manager is also an employee), and the $\langle \text{Eid}, \text{Mid} \rangle$ pairs define a management tree.

For $s < 1$, UpSizeR can scale T by selecting a subtree of appropriate size; for $s = 1$, it can iteratively and randomly swap subtrees.

For $s > 1$, UpSizeR can first break the loop by removing **Mid** from T and introducing a temporary table T' with primary key **E** and foreign key **Mid** that references **Eid** in T . After running UpSizeR on this acyclic schema, the **Mid** column from T' is inserted into T ; to ensure the tree property, **Eid** and **Mid** can be sorted, so that $\text{Eid} \geq \text{Mid}$ in every row.

The UpSizeR user can also supply some algorithm $\mathcal{A}$ for generating application-specific trees of specified size. In that case, UpSizeR first runs $\mathcal{A}$ to generate the $\langle \text{Eid}, \text{Mid} \rangle$ pairs for T , then use the algorithms in Sec. 3.5 to replicate correlation between **Mid** and foreign keys in T and other tables.

4.4 Nonstatic degree distribution

UpSizeR first scales by s all tables in $\mathcal{D}_0$ (see Sec. 3.4), and the other table sizes are then indirectly determined by the degree distributions. It follows from the static degree distribution assumption (A4) that all tables are scaled by s approximately. This may not be the right thing to do.

Some datasets may have fixed-size tables (e.g. **NATION** in TPC-H), thus changing the degree distribution in $\tilde{\mathcal{D}}$. Suppose T has a primary key that appears as foreign key in T' . If T is static, then UpSizeR uses $s \times \deg(v, T')$ for scaling. If T' is static, then UpSizeR can use $\deg(v, T')/s$ but, in practice, it is unlikely that a foreign key is unaffected by a scale up in the primary key.

Another reason for a change in degree distribution when a dataset grows lies in the passage of time. For example, one expects the number of comments posted by a user to increase over time, possibly shifting the $\deg(\text{UId}, \text{Photo})$ distribution. We can further relax (A4) by having UpSizeR extract the degree growth function by mining the dates in the data (e.g. **Cdate** in **Comment**, **Tdate** in **Tag**, etc.).

4.5 Content generation for non-key attributes

This paper focuses on replicating key value correlation. There are many ways of generating non-key attribute values.

For example, one could extract from $\mathcal{D}$ relevant probability distributions (e.g. for **Ulocation**, **Psize** or **Clanguage**) and use them for content generation. The UpSizeR user can also supply a method for, say, generating fake **Comment** text (e.g. **TEXTURE** [11]).

Non-key attributes may also induce tuple correlation, thus violating assumption (A5). For example, there is some distribution for how many photographs an $\mathcal{F}$ user uploads per day. UpSizeR can replicate such correlation with $\deg(d, \text{Photo})$ where d is a value for **Pdate** (which is not a key).

A more difficult form of tuple correlation for a non-key attribute in $\mathcal{F}$ is tag value (“bird”, “car”, etc.). The tags used by a bird watcher are likely to have a coherence that makes them recognizably different from those used by a car enthusiast. One must take such coherence into account when generating synthetic tags.

For the experiments in Sec. 6, we use the following algorithm to generate the taglist for a user $\tilde{u}$ in $\tilde{\mathcal{F}}$, by perturbing the taglist ℓ of a random real user in $\mathcal{F}$: determine $\|\ell_{\tilde{u}}\|$, i.e. how many tags $\tilde{u}$ uses, from **Tag** in $\tilde{\mathcal{F}}$; if $\|\ell\| > \|\ell_{\tilde{u}}\|$, randomly remove excess tags from ℓ ; else if $\|\ell\| < \|\ell_{\tilde{u}}\|$, use the joint tag distribution to add tags to ℓ ; we thus get $\|\ell\| = \|\ell_{\tilde{u}}\|$, and can assign ℓ to $\ell_{\tilde{u}}$.

Photo			Comment			
Pid	Puid	...	Cid	CPid	Cuid	...
P_x	x			P_x	y	
P_y	y			P_y	x	

Figure 4: Users x and y comment on each other’s photographs. Such interactions induce inter-column and inter-row correlations in the tables above.

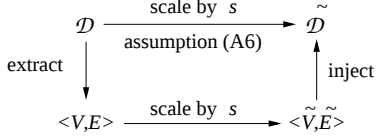


Figure 5: How UpSizeR can replicate correlation in a social network dataset $\mathcal{D}$ by extracting and scaling the social interaction graph $\langle V, E \rangle$.

5. LIMITATIONS

The main issue in scaling an empirical dataset lies in the correlations, but it is computationally intractable to replicate all of them. For example, extracting pairwise correlation in tag values from $\mathcal{F}$ (Sec. 4.5) takes far longer than the rest of UpSizeR. One must therefore judiciously choose which correlations to replicate. This choice can be made by the UpSizeR user, or by examining the queries (Sec. 7).

While the algorithms described so far may suffice for classical commercial datasets (in banking, telecom, etc.), social network data require more. For example, Flickr friends are more likely to comment on each other’s photographs. Such an interaction appears in $\mathcal{F}$ as inter-column and inter-row correlation as illustrated in Fig. 4. Such correlations that are induced by social interactions go beyond assumption (A6). How can UpSizeR replicate such correlations?

Fig. 5 illustrates one possibility: First extract a graph $\langle V, E \rangle$ from $\mathcal{D}$, where the nodes in V represent members of the social network, and each edge in E represents a social interaction. This graph is then scaled by s to $\langle \tilde{V}, \tilde{E} \rangle$, and “injected” into $\tilde{\mathcal{D}}$ (constructed under assumption (A6)) by modifying the values in $\tilde{\mathcal{D}}$.

In scaling $\langle V, E \rangle$, UpSizeR must replicate the topology — number of triangles (a friend of a friend is likely to be a friend), path lengths (6 degrees of separation), etc.; this is non-trivial, but graph-theoretic. Graph extraction and injection, however, would require a database-theoretic understanding of social networks. Given a relational dataset $\tilde{\mathcal{D}}$ from a social network, how does one extract a social interaction graph? Conversely, what data dependencies are induced in the relations by, say, the triangles in the graph? We state this issue as the **Social Networks’ Attribute Correlation Problem**:

Suppose a relational database state $\mathcal{D}$ records data from a social network. How do the social interactions affect the correlation among attribute values in $\mathcal{D}$ ’s tables?

There are many papers on online social networks, but we found none that translates that literature into relational database theory. We believe this Attribute Correlation Problem points to a rich, new area for database research.

6. EXPERIMENTS

The TPC benchmarks are well-established, so the reader may expect us to compare UpSizeR to one of them; on the other hand, TPC datasets are purely synthetic. We therefore postpone a comparison to Sec. 10.1 in the Appendix.

To validate UpSizeR, we need to compare its results against real datasets for various values of s . However, we have no access to any real commercial data from, say, a bank or retailer. We therefore use crawled data from Flickr for comparison in Sec. 6.1 below.

One motivation for UpSizeR is in scalability testing, so we also validate UpSizeR with a scale out test in Sec. 6.2.

6.1 UpSizeR Validation with Flickr

We downloaded four datasets from Flickr for $\mathcal{F}$. These datasets were then combined to give different sizes.

The downloads were at different times. Since $\deg(x, \text{Photo})$, $\deg(x, \text{Comment})$ and $\deg(x, \text{Tag})$ generally increase over time for any user x , the static degree assumption (A4) does not hold. Although we can extend UpSizeR to model this effect of time (Sec. 4.4), we impose (A4) in this validation exercise by keeping each pair of datasets disjoint through renaming. In other words, if two downloaded datasets $\mathcal{E}_1$ and $\mathcal{E}_2$ have some common Uids (say), we rename the Uids in one of them so that $\mathcal{E}_1$ and $\mathcal{E}_2$ have no common Uids.

Rather than try to control the scale factor for the real datasets, we let them decide the s value for UpSizeR. Specifically, since the scaling up starts with $\mathcal{D}_0 = \{\text{User}\}$, we obtain s by $s = t_1/t_2$, where t_i is the number of Uids in an $\mathcal{F}$ dataset. The baseline size is given by a fixed dataset $\mathcal{F}_{1.00}$ and, in general, $\mathcal{F}$ datasets are denoted $\mathcal{F}_s$ according to their s value when compared to $\mathcal{F}_{1.00}$. For example, $\mathcal{F}_{2.81}$ has a number of Uids that is 2.81 times that in $\mathcal{F}_{1.00}$.

Hence, the validation is a comparison between a real $\mathcal{F}_s$ and a synthetic UpSizeR($\mathcal{F}_{1.00}, s$), as shown in Table 1

Since the s value given to UpSizeR is calculated from the real dataset $\mathcal{F}_s$, and the scaling up starts with $\mathcal{D}_0 = \{\text{User}\}$, the close agreement in Table 1 between real and synthetic User is expected.

For $\mathcal{D}_1 = \{\text{Photo}\}$, UpSizeR uses the degree distribution to generate the table. The efficacy of doing so cannot be taken for granted, since the difference between $\mathcal{F}_{1.00}$ and UpSizeR($\mathcal{F}_{1.00}, 1.00$) is about 10% for Photo in Table 1. The agreement is better for the other s values.

For $\mathcal{D}_2 = \{\text{Comment}, \text{Tag}\}$, Table 1 shows that UpSizeR’s use of co-clustering produces table sizes that are in good agreement, considering the vagaries of real data in $\mathcal{F}_s$.

However, merely matching table sizes does not suffice. We have argued that dataset similarity should be judged with queries. Since the UpSizeR version in this paper does not replicate social interactions, we avoid querying the social network below. We start with the following queries:

- F1:** Retrieve users who uploaded photographs (0 joins).
- F2:** Retrieve photographs that are commented on by their owners (1 join).
- F3:** Retrieve users who tagged others’ photographs (1 join).
- F4:** Retrieve users who uploaded photographs but made no comments (2 joins).

F1 tests UpSizeR’s ability to reproduce selectivity, while F2, F3 and F4 test for correct join sizes. The number of tuples retrieved by these queries for the real $\mathcal{F}_s$ and UpSizeR($\mathcal{F}_{1.00}, s$) are also shown in Table 1.

Agreement in query results is harder to achieve than table

#tuples	User	Photo	Comment	Tag	F1	F2	F3	F4	F5	F6
$\mathcal{F}_{1.00}$	146374	529926	1505267	3343964	945	85137	2654	1	2075	120
UpSizeR($\mathcal{F}_{1.00}, 1.00$)	146374	581069	1654678	3765474	906	71080	2896	0	3081	161
$\mathcal{F}_{2.81}$	410892	1557856	4234147	9198476	2398	219499	9717	3	8448	255
UpSizeR($\mathcal{F}_{1.00}, 2.81$)	411305	1557650	4410086	10377427	2687	205334	8119	1	9973	474
$\mathcal{F}_{5.35}$	783821	2803603	7709470	16299952	4369	401464	15671	4	15513	485
UpSizeR($\mathcal{F}_{1.00}, 5.35$)	783090	2823268	8093519	17813587	5063	406099	15751	5	17306	972
$\mathcal{F}_{9.11}$	1332796	4474956	18136861	27743408	8258	734766	27491	15	32619	1513
UpSizeR($\mathcal{F}_{1.00}, 9.11$)	1333448	4693496	13702306	29637029	8673	717454	26686	13	31640	1746

Table 1: Comparing table sizes and query results for real $\mathcal{F}_s$ and synthetic UpSizeR($\mathcal{F}_{1.00}, s$).

sizes, since the datasets must also have similar data correlation. It is hence not surprising that discrepancy is now larger. Still, agreement is generally good. Note in particular that the UpSizeR datasets accurately return a very small number of tuples for F4 from among the millions of tuples for photographs and comments.

Although F3 tests the join of **Tag** and **Photo**, it does not query the (non-key) tag values. To test the taglist generation described in Sec. 4.5, we use the following:

F5: Retrieve photographs tagged with “bird”.

F6: Retrieve photographs tagged with “bird” and “sky”.

In particular, F6 tests our algorithm for replicating tag coherence.

Table 1 shows that the number of tuples retrieved by F5 and F6 from the synthetic datasets, although inaccurate, are in the right ball park. Surely, there is room for improvement (e.g. associate “bird” and “birds”), but the F5 and F6 results already show how inter-tuple correlation induced by non-key attributes can be captured through content generation, like what we have done through taglist generation by sampling and perturbation.

6.2 UpSizeR Validation with Scale Out Test

Every system must encounter, sooner or later, some hardware or software bottleneck that causes performance to saturate. Beyond the saturation point, resource contention usually causes inefficiencies that result in performance degradation. We now validate UpSizeR’s accuracy in locating where performance begins to degrade as a system is scaled out.

For this experiment, we include synthetic blobs (binary large objects) in place of photographs. These blobs are stored in Hadoop Object Store [19], a storage system — similar to Facebook’s custom-built Haystack [1] — that we built on top of Hadoop.

We use the same cluster as in the TPC-H experiments (Sec. 10.1). We run C concurrent queries, where each query retrieves all photographs uploaded by a user who is randomly selected by name. (On average, each user in our Flickr data uploads 550 photographs.) We fix C by replacing each terminating query with a new query.

For each s , there is a concurrency C_s where throughput is maximum: as C increases, throughput is increasing for $C < C_s$ and decreasing for $C > C_s$. Our experiment tests if UpSizeR data can accurately predict the saturation point C_s for each s .

For validation, we use the real non-blob data for $s = 1.00, 2.81, 5.35$ and 9.11 . With the blobs included, the total dataset sizes are approximately 50GBytes for $\mathcal{F}_{1.00}$, 132GBytes for $\mathcal{F}_{2.81}$, 271GBytes for $\mathcal{F}_{5.35}$ and 426GBytes for $\mathcal{F}_{9.11}$.

We are limited by the size of our machine cluster, so the

number of nodes used in the scale out test is set to 2, 6, 10 and 18 for $s = 1.00, 2.81, 5.35$ and 9.11 respectively.

Table 2 shows that throughput (queries per second) measured when the queries run on UpSizeR data is close to that for real Flickr non-blob data. More important, the synthetic dataset correctly identifies the concurrency C_s at which throughput begins to degrade.

One would hope that, as a system scales out, it can support higher concurrency; in particular, C_s should scale linearly with s . However, for this workload, Table 2 shows $C_{1.00} = 10$, $C_{2.81} = 24$, $C_{5.35} = 24$ and $C_{9.11} = 28$. Such a prediction of weak scalability with UpSizeR data would suggest a need for performance debugging or system redesign.

7. RELATED WORK (CONTINUED)

Bruno and Chaudhuri’s Data Generation Language [5] can specify value distributions and generate data tuples, while Hoag and Thompson’s Synthetic Data Description Language [12] has a construct for specifying foreign keys, but data generation by both languages do not replicate correlation between foreign keys (like UpSizeR does with joint degree and co-clustering distributions).

Houkjaer et al. have presented a data generating tool [13] that processes the schema graph in the opposite order to UpSizeR. (For the Flickr example of Fig. 2, they would generate **Comment** and **Tag** before **Photo** and **User**.) This necessitates the use of temporary foreign key values that are replaced after primary key values are generated later. The tool also uses cardinalities and value (not correlation) distributions extracted from real data.

CORDS is a tool that uses the application queries to select columns whose correlations are important for query optimization [14]. It also uses the correlation to generate synthetic data, but this purely valued-based generation is different from the entity-based (i.e. key-based) generation underlying UpSizeR. CORADD is another tool that discovers attribute correlations that are important to the queries, and use them to design materialized views and indexes [15]. UpSizeR can follow CORDS and CORADD in using the queries to select the attributes for correlation replication.

8. CONCLUSION

We have presented UpSizeR, a software that addresses the Dataset Scaling Problem. UpSizeR’s ability to synthetically scale an empirical dataset makes it a tool for generating application-specific benchmarks.

Table 1 confirms that UpSizeR can accurately scale up table sizes for the Flickr dataset, and the query results show good agreement with crawled data. Table 2 also shows that

concurrency C	1	4	8	10	12	16	20		
$\mathcal{F}_{1.00}$	0.040	0.125	0.172	<u>0.186</u>	0.175	0.167	0.142		
UpSizeR($\mathcal{F}_{1.00}$, 1.00)	0.043	0.131	0.164	<u>0.185</u>	0.169	0.167	0.145		
concurrency C	1	4	8	12	16	20	24	28	32
$\mathcal{F}_{2.81}$	0.039	0.153	0.271	0.312	0.339	0.355	<u>0.378</u>	0.358	0.356
UpSizeR($\mathcal{F}_{1.00}$, 2.81)	0.041	0.147	0.273	0.304	0.348	0.362	<u>0.389</u>	0.355	0.348
concurrency C	1	4	8	12	16	20	24	28	32
$\mathcal{F}_{5.35}$	0.038	0.149	0.253	0.276	0.313	0.326	<u>0.354</u>	0.342	0.336
UpSizeR($\mathcal{F}_{1.00}$, 5.38)	0.039	0.141	0.273	0.301	0.327	0.333	<u>0.359</u>	0.347	0.341
concurrency C	1	4	8	12	16	20	24	28	32
$\mathcal{F}_{9.11}$	0.040	0.154	0.296	0.323	0.377	0.384	0.401	<u>0.422</u>	0.418
UpSizeR($\mathcal{F}_{1.00}$, 9.11)	0.043	0.154	0.275	0.343	0.354	0.386	0.423	<u>0.435</u>	0.424

Table 2: A scale out test that compares throughput (queries per second) when concurrent queries retrieve blobs from the real and synthetic datasets. The underlined numbers indicate where throughputs are maximum; note that the synthetic data accurately predict where throughput begins to degrade.

the datasets produced by UpSizeR can accurately predict where throughput degrades when scaling out a system.

These results suggest that UpSizeR has the basic techniques for scaling up traditional datasets (in retail, logistics, etc.). Indeed, Sec. 10.1 in the Appendix shows that UpSizeR can match TPC-H query results and response time.

Given the proliferation of relational databases, and the heterogeneity among them, an attack on the Dataset Scaling Problem is now overdue. UpSizeR is only a first-cut solution, and much remains to be done. We have therefore released UpSizeR for open-source development by the database community.

Currently, UpSizeR does not replicate correlations induced by social interactions (so Sec. 6.1 avoids querying the social network). The extension sketched in Fig. 5 would require graph-theoretic techniques that properly scale the degrees, triangles, paths, etc. It also requires a database-theoretic understanding of inter-column and inter-row correlations generated by social interactions (a probabilistic dependency theory for the Attribute Correlation Problem, say); we see this as a very promising area for future work.

9. REFERENCES

- [1] D. Beaver, S. Kumar, H. C. Li, et al. Finding a needle in Haystack: Facebook’s photo storage. In *OSDI*, 2010.
- [2] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the weather tomorrow?: towards a benchmark for the cloud. In *DBTest*, pages 1–6, 2009.
- [3] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *ICDE*, pages 506–515, 2007.
- [4] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. QAGen: generating query-aware test databases. In *SIGMOD*, pages 341–352, 2007.
- [5] N. Bruno and S. Chaudhuri. Flexible database generators. In *VLDB*, pages 1097–1107, 2005.
- [6] M. Castellanos, B. Zhang, I. Jimenez, et al. Data desensitization of customer data for use in optimizer performance experiments. In *ICDE*, pages 1081–1092, 2010.
- [7] R. Chaiken, B. Jenkins, P.-A. Larson, et al. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.
- [8] B. F. Cooper, A. Silberstein, E. Tam, et al. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.
- [9] D. J. DeWitt. The Wisconsin Benchmark: Past, present, and future. In *The Benchmark Handbook*. Morgan Kaufmann Publishers Inc., 1993.
- [10] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *KDD*, pages 89–98, 2003.
- [11] V. Ercegovac, D. J. DeWitt, and R. Ramakrishnan. The TEXTURE benchmark: measuring performance of text queries on a relational DBMS. In *VLDB*, pages 313–324, 2005.
- [12] J. E. Hoag and C. W. Thompson. A parallel general-purpose synthetic data generator. *SIGMOD Rec.*, 36(1):19–24, 2007.
- [13] K. Houkjaer, K. Torp, and R. Wind. Simple and realistic data generation. In *VLDB*, pages 1243–1246, 2006.
- [14] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulmaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658, 2004.
- [15] H. Kimura, G. Huo, A. Rasin, S. u. Madden, and S. Zdonik. CORADD: Correlation aware database designer for materialized views and indexes. In *PVLDB*, 2010.
- [16] M. E. Nergiz, C. Clifton, and A. E. Nergiz. Multirelational k-anonymity. *IEEE Trans. on Knowl. and Data Eng.*, 21(8):1104–1117, 2009.
- [17] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang. The case for application-specific benchmarking. In *HOTOS*, pages 102–109, 1999.
- [18] J. M. Stephens and M. Poess. MUDD: a multi-dimensional data generator. In *WOSP*, pages 104–109, 2004.
- [19] D. T. Wang. Data storage and retrieval for social network services. Master’s thesis, National University of Singapore, Singapore, June 2010.
- [20] R. Yahalom, E. Shmueli, and T. Zrihen. Constrained anonymization of production data: a constraint satisfaction problem approach. In *SDM*, pages 41–53, 2010.

10. APPENDIX

This appendix presents more experimental results and UpSizeR details. Sec. 10.1 compares results of queries that are run on data generated by DBGen and UpSizeR for the TPC-H schema. Sec. 10.2 further describes UpSizeR in pseudocode.

10.1 Comparing UpSizeR and TPC-H

One can view UpSizeR as generating datasets for benchmarking. The TPC benchmarks are *de rigueur* for database systems, and they also generate datasets to specified scale. Since the TPC benchmarks are widely accepted, the reader may reasonably expect a comparison between UpSizeR and a TPC benchmark. We do that here, and also demonstrate the use of UpSizeR for scaling down.

The latest TPC benchmarks are TPC-E for online transaction processing and TPC-H for decision support. TPC-E has tables that contain three or more foreign keys, thus requiring the use of compute-intensive high-dimensional co-clustering. We therefore choose to validate with TPC-H.

TPC-H datasets are generated by DBGen and specified by size. The 1GB, 2GB, 10GB and 40GB DBGen datasets are denoted $\mathcal{H}_1$, $\mathcal{H}_2$, $\mathcal{H}_{10}$ and $\mathcal{H}_{40}$, respectively. We use UpSizeR to scale down $\mathcal{H}_{40}$ with $s = 0.025, 0.05$ and 0.25 . Thus, $\text{UpSizeR}(\mathcal{H}_{40}, 0.025)$ is a dataset that is similar in size to $\mathcal{H}_1$, and replicates data correlation extracted from $\mathcal{H}_{40}$.

The correlation in DBGen data lies in the key values, and this correlation is replicated by UpSizeR through the use of joint degree and co-clustering distributions. For the generation of non-key values, UpSizeR uses the same techniques as DBGen when scaling down $\mathcal{H}_{40}$.

We ran UpSizeR on a database server with a 64GB RAM. This put a constraint on the input size $\mathcal{D}$ for UpSizeR, which is why we did not scale down from a bigger $\mathcal{H}_s$.

The queries we use to compare DBGen data and UpSizeR output are simplified versions of TPC-H queries, as shown in Fig. 7. The comparison is in terms of number of tuples retrieved, the aggregates computed, and the response time.

The run times are measured on a cluster of 14 commodity machines, each having an Intel Xeon Quad Core, (2.4GHz, 8MB cache), 2×4GB ECC RAM and 2×500GB hard disk (SATA II, 3.5 inches). They run 64-bit Linux CentOS (release 5.5), the files are stored with Hadoop 0.20.2 and the queries executed with Hive 0.5.0. (Hadoop⁵ is an open-source variant of MapReduce; Hive⁶ is a data warehouse infrastructure built on Hadoop.) We use default values for the parameters (e.g. 3 replicas).

Table 3 shows good agreement in the number of tuples returned by the queries. Table 3 shows weaker agreement for response times. We expected this since even the same dataset, when loaded into Hive on separate occasions, can generate different query response times.

Query H1 computes `ave()` and `count()` and H4 computes `sum()`, so the appropriate comparison is in the returned values. Table 4 shows that the aggregates computed with UpSizeR output agrees well with those from DBGen.

10.2 UpSizeR’s pseudocode

This section presents pseudocode for the UpSizeR algorithm in Sec. 3, *without* the extensions in Sec. 4.

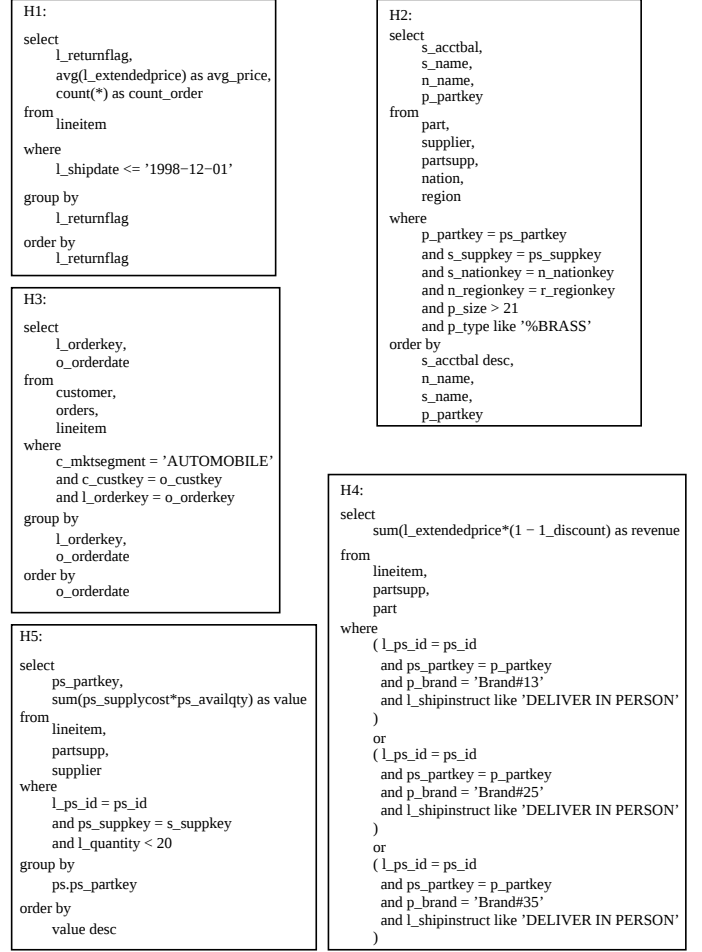


Figure 7: Queries used to compare DBGen data and UpSizeR output.

Algo. 1: $\text{UpSizeR}(\mathcal{D}, x)$ starts with the extraction of probability distributions (Sec. 3.1), sorting of tables into $\mathcal{D}_i$ (Sec. 3.2), partitioning of $\mathcal{D}_i$ into equivalence classes $[\mathbf{K}]$ (Sec. 3.3), and table generation for $\mathcal{D}_0$ (Sec. 3.4 and Algo. 2: $\text{gen0FKTable}(T, s)$). Algo. 1 then enters the main loop (Sec. 3.5).

To generate a table $\tilde{T}'$ with foreign key(s), Algo. 3: $\text{genTables}(\mathbf{K})$ first generates $\text{deg}(\tilde{v}, \tilde{T}')$ (Sec. 3.5.1 and Algo. 4: $\text{genDegree}(\tilde{v}, [\mathbf{K}])$). The two cases in Sec. 3.5.2 correspond to Algo. 5: $\text{gen1FKTable}(K)$ and Algo. 6: $\text{gen2FKTable}(K_1, K_2)$.

⁵<http://hadoop.apache.org/>

⁶<http://hadoop.apache.org/hive/>

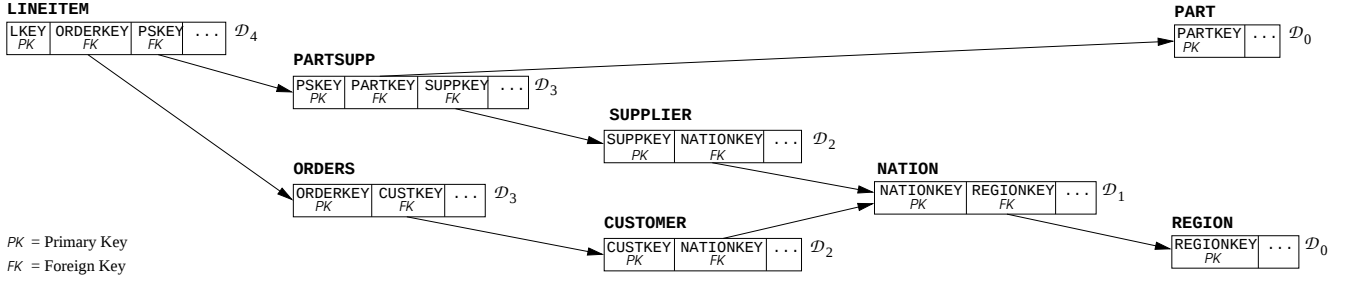


Figure 6: Schema $\mathcal{H}$ for the TPC-H benchmark that is used for UpSizeR comparison in Sec. 10.1. Sorting this graph into $\mathcal{D}_i$ (Sec. 3) gives $\mathcal{D}_0 = \{\text{PART}, \text{REGION}\}$, $\mathcal{D}_1 = \{\text{NATION}\}$, $\mathcal{D}_2 = \{\text{SUPPLIER}, \text{CUSTOMER}\}$, $\mathcal{D}_3 = \{\text{PARTSUPP}, \text{ORDERS}\}$ and $\mathcal{D}_4 = \{\text{LINEITEM}\}$. $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_4$ have one equivalence class each, namely $[\{\text{REGIONKEY}\}] = \{\text{NATION}\}$ for $\mathcal{D}_1$, $[\{\text{NATIONKEY}\}] = \{\text{SUPPLIER}, \text{CUSTOMER}\}$ for $\mathcal{D}_2$ and $[\{\text{PSKEY}, \text{ORDERKEY}\}] = \{\text{LINEITEM}\}$ for $\mathcal{D}_4$. However, $\mathcal{D}_3$ has two equivalence classes: $[\{\text{PARTKEY}, \text{SUPPKEY}\}] = \{\text{PARTSUPP}\}$ and $[\{\text{CUSTKEY}\}] = \{\text{ORDERS}\}$.

	#tuples(time)	H1	H2	H3	H4	H5
1GB	DBGen $\mathcal{H}_1$	3 (52s)	92196(192s)	297453(173s)	1(138s)	199998(135s)
	UpSizeR($\mathcal{H}_{40}, 0.025$)	3 (50s)	92667(194s)	302264(177s)	1(150s)	199999(141s)
2GB	DBGen $\mathcal{H}_2$	3 (77s)	184156(203s)	597099(206s)	1(188s)	399995(156s)
	UpSizeR($\mathcal{H}_{40}, 0.05$)	3 (78s)	185693(196s)	595398(212s)	1(188s)	399998(155s)
10GB	DBGen $\mathcal{H}_{10}$	3(131s)	927140(265s)	3000540(297s)	1(356s)	1999983(235s)
	UpSizeR($\mathcal{H}_{40}, 0.25$)	3(168s)	928464(280s)	3006459(352s)	1(346s)	1999980(239s)

Table 3: A comparison of resulting number of tuples and execution time (in brackets) when the queries H1,...,H5 in Fig. 7 are run over TPC-H data generated with DBGen and UpSizeR. Table 4 shows the aggregates computed by H1 and H4.

Algorithm 1 UpSizeR($\mathcal{D}, s$)

Input: database state $\mathcal{D}$ and a scale factor s
Output: a synthetic database state that scales up $\mathcal{D}$ by s

```

get joint degree distribution  $f_K$  from  $\mathcal{D}$  for each key  $K$ 
get co-clustering distribution  $f_T^{cc}$  for each table  $T$ 
use the schema graph to sort  $\mathcal{D}$  into  $\mathcal{D}_0, \mathcal{D}_1, \dots$ 
partition each  $\mathcal{D}_i$  into equivalence classes  $[\mathbf{K}]$ 
for all  $T \in \mathcal{D}_0$  do
    gen0FKTable( $T, s$ )
end for
i  $\leftarrow 0$ 
repeat
    i  $\leftarrow i + 1$ 
    for all  $T \in \mathcal{D}_i$  do
        flag( $T$ )  $\leftarrow$  false
    end for
    for all  $T \in \mathcal{D}_i$  and flag( $T$ ) = false do
        Let  $\mathbf{K}$  be the set of foreign keys in  $T$ 
        genTables( $\mathbf{K}$ )
        for all  $T' \in [\mathbf{K}]$  do
            flag( $T'$ )  $\leftarrow$  true
        end for
    end for
until all tables are generated

```

Algorithm 2 gen0FKTable(T, s)

Input: table T with no foreign keys and scale factor s
Output: a synthetic $\tilde{T}$ that is s times the size of T

```

let  $t$  be the number of  $T$  tuples in the given  $\mathcal{D}$ 
for  $i = 1$  to  $st$  do
    generate a unique primary key value  $\tilde{v}$ 
    genContent( $\tilde{T}, \tilde{v}$ )
end for

```

Algorithm 3 genTables($\mathbf{K}$)

Input: a set of keys $\mathbf{K}$
Output: a synthetic $\tilde{T}$ for each $T \in [\mathbf{K}]$

```

for all  $K \in \mathbf{K}$  do
    for all  $K$  value  $\tilde{v}$  do
        genDegree( $\tilde{v}, [\mathbf{K}]$ )
    end for
end for
if  $\mathbf{K} = \{K\}$  then
    gen1FKTable( $K$ )
end if
if  $\mathbf{K} = \{K_1, K_2\}$  then
    gen2FKTable( $K$ )
end if

```

		H1 avg(count)			H4
	l_returnflag	A	N	R	sum
1GB	DBGen $\mathcal{H}_1$	38273 (1478493)	38248 (3043852)	38250 (1478870)	6.586E09
	UpSizeR($\mathcal{H}_{40}, 0.025$)	38252 (1482196)	38246 (3042663)	38216 (1483192)	6.550E09
2GB	DBGen $\mathcal{H}_2$	38252 (2959267)	38234 (6076312)	38263 (2962417)	1.306E10
	UpSizeR($\mathcal{H}_{40}, 0.05$)	38246 (2963035)	38245 (6083163)	38239 (2965648)	1.304E10
10GB	DBGen $\mathcal{H}_{10}$	38237(14804077)	38234(30373792)	38251(14808183)	6.554E10
	UpSizeR($\mathcal{H}_{40}, 0.25$)	38236(14802818)	38237(30387309)	38243(14808265)	6.556E10

Table 4: A comparison of returned aggregate values: ave() and count() for H1, sum() for H4. (A, N and R are values of l_returnflag.)

Algorithm 4 genDegree($\tilde{v}, [\mathbf{K}]$)

Input: a value $\tilde{v}$ for key $K \in \mathbf{K}$ where
 $[\mathbf{K}]$ is the set of tables with $\mathbf{K}$ as foreign keys

Output: $deg(\tilde{v}, \tilde{T}')$ for each $\tilde{T} \in [\mathbf{K}]$

let $T_1'', T_2'', \dots, T_m''$ be the tables for which
earlier calls on genDegree has generated
 $deg(\tilde{x}, \tilde{T}_1''), \dots, deg(\tilde{x}, \tilde{T}_m'')$ for all K value $\tilde{x}$

let $[\mathbf{K}] = \{T_1', \dots, T_n'\}$

derive from the joint degree distribution f_K
the conditional distribution

$\Pr(deg(\tilde{v}, T_1') = d_1', \dots, deg(\tilde{v}, T_n') = d_n'$
 $\mid deg(\tilde{v}, T_1'') = d_1, \dots, deg(\tilde{v}, T_m'') = d_m)$

for all T_i' **do**

use the conditional distribution to choose $deg(\tilde{v}, \tilde{T}_i')$

end for

Algorithm 5 gen1FKTable(K)

Input: primary key K

Output: a table for each $T' \in [\{K\}]$

for all $T' \in [\{K\}]$ **do**

for all K value $\tilde{v}$ **do**

choose $deg(\tilde{v}, \tilde{T}')$ according to degree distribution

end for

repeat

generate a new value $\tilde{v}'$ for primary key of T'

choose $\tilde{v}$ with probability proportional to $deg(\tilde{v}, \tilde{T}')$

decrement $deg(\tilde{v}, \tilde{T}')$

genContent($\tilde{T}', \tilde{v}'$)

until $\sum_{\tilde{v}} deg(\tilde{v}, \tilde{T}') = 0$

end for

Algorithm 6 gen2FKTable(K_1, K_2)

Input: primary keys K_1 and K_2

Output: a table for each $T' \in [\{K_1, K_2\}]$

for all $T' \in [\{K_1, K_2\}]$ **do**

for all K_1 value $\tilde{v}_1$ and K_2 value $\tilde{v}_2$ **do**

assign $\tilde{v}_1$ and $\tilde{v}_2$ to clusters cc_1 and cc_2

according to marginal distributions of $f_{T'}^{cc}$

randomly choose $deg(\tilde{v}_1, \tilde{T}')$ and $deg(\tilde{v}_2, \tilde{T}')$

according to the degree distributions

end for

make $\sum_{\tilde{v}_1} deg(\tilde{v}_1, \tilde{T}')$ and $\sum_{\tilde{v}_2} deg(\tilde{v}_2, \tilde{T}')$ equal

repeat

generate a new value $\tilde{v}'$ for primary key of T'

assign $\tilde{v}'$ to a random (cc_1, cc_2) according to $f_{T'}^{cc}$

randomly choose $\tilde{v}_1$ in cc_1 with probability

proportional to $deg(\tilde{v}_1, \tilde{T}')$

randomly choose $\tilde{v}_2$ in cc_2 with probability

proportional to $deg(\tilde{v}_2, \tilde{T}')$

create a $\tilde{T}'$ tuple with key values $\tilde{v}', \tilde{v}_1', \tilde{v}_2'$

for primary key, K_1 and K_2

decrement $deg(\tilde{v}_1, \tilde{T}')$ and $deg(\tilde{v}_2, \tilde{T}')$

genContent($\tilde{T}', \tilde{v}'$)

until $\sum_{\tilde{v}_1} deg(\tilde{v}, \tilde{T}') = 0 = \sum_{\tilde{v}_2} deg(\tilde{v}_2, \tilde{T}')$

end for

Algorithm 7 genContent($\tilde{T}', \tilde{v}'$)

Input: a synthetic table $\tilde{T}'$ and key value $\tilde{v}'$

Output: tuple for $\tilde{v}'$ acquires non-key values

for an attribute A that is not a primary or foreign key,
this paper assumes that A 's value is a function of the
key values, and is generated through sampling from $\mathcal{D}$,
data mining or user-supplied methods etc.
