Distributing Data for Secure Database Services

Vignesh Ganapathy
Google Inc.
vignesh@google.com

Dilys Thomas
Stanford University
dilys@cs.stanford.edu

Tomas Feder
Stanford University
tomas@theory.stanford.edu

Hector Garcia-Molina
Stanford University
hector@cs.stanford.edu

Rajeev Motwani
Stanford University
rajeev@cs.stanford.edu

Abstract

The advent of database services has resulted in privacy concerns on the part of the client storing data with third party database service providers. Previous approaches to enabling such a service have been based on data encryption, causing a large overhead in query processing. A distributed architecture for secure database services is proposed as a solution to this problem where data was stored at multiple sites. The distributed architecture provides both privacy as well as fault tolerance to the client. In this paper we provide algorithms for (1)distributing data: our results include hardness of approximation results and hence a heuristic greedy hill climbing algorithm for the distribution problem (2) partitioning the query at the client to queries for the various sites is done by a bottom up state based algorithm we provide. Finally the results at the sites are integrated to obtain the answer at the client. We provide an experimental validation and performance study of our algorithms .

1. Introduction

Database service providers are becoming ubiquitous these days. These are companies which have the necessary hardware and software setup (data centers) for storage and retrieval of terabytes of data [6],[9],[20]. As a result of such service providers, parties wanting to store and manage their data may prefer to outsource data to these service providers. The parties who outsource their data will be referred to as *clients* hereafter. The service providers storing data will be referred to as *servers*.

There is a growing concern regarding data privacy among clients. Often, client data has sensitive information which they do not want to compromise. Examples of sensitive databases include a payroll database or a medical database. To capture the notions of privacy in a database, privacy constraints are specified by the client on the columns of the sensitive database. We use the notion of privacy constraints as described in [2], [18]. An example of a privacy constraint is (age, salary) which states that age and salary columns of a tuple must not be accessible together at the servers. The clients also have a set of queries also known as the workload that need to executed on a regular basis on their outsourced database.

Some of the existing solutions for data privacy involve encryption of the entire database when storing it on the server. A client query request requires the entire database to be transferred to the client and decrypted to get the result. However, this has the disadvantage of heavy network traffic as well as decryption cost for each query. One solution is to encrypt sensitive columns only instead of the entire database so many queries do not require the entire relation to be transmitted to the client.

Reference [2] suggests using two (multiple) service providers in order to store the data. The advantage of using two servers is that the columns can be split across the two servers to satisfy privacy constraints without encrypting the split columns. Thus, in order to satisfy privacy constraints, columns can either be split across servers or stored encrypted. Thus the goal of any decomposition algorithm is to partition the database to satisfy the following.

1. None of the privacy constraints should be violated.
2. For a given workload, minimum number of bytes should be transferred from the servers to the client.

We explain both of the above points in detail in the next section. The problem of finding the optimal partition structure for a given set of privacy constraints and query workload can be shown to be intractable. We apply heuristic search techniques based on Greedy Hill Climbing to come up with nearly optimal solutions.

Layout: The next section explains relevant terminology and system architecture for such a database service. The intractability of the problem is described in detail in Section 3. The hill climbing technique is then discussed in detail followed by experimental setup and results. We then conclude with related and future work in this direction.

2 System Architecture

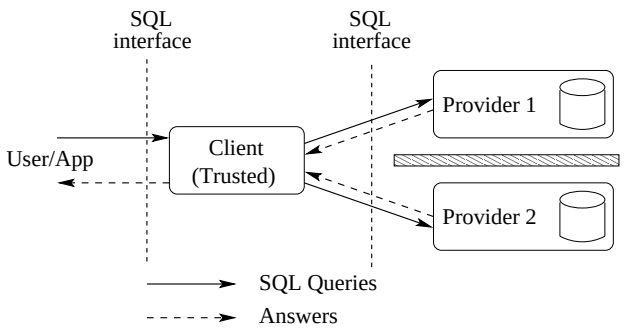

Figure 1. Distributed Architecture for a Secure Database Service

The general architecture of a distributed secure database service, as illustrated in Figure 1 is described more in [2]. It consists of a trusted client as well as two or more servers that provide a database service. The servers provide reliable content storage and data management but are not trusted by the client to preserve content privacy.

Some relevant terms are described here before going into further details.

- **Data Schema** This is the schema of the relation the client wishes to store on the server. As a running example, consider a company desiring to store relation R with the following schema.
  `R (Name, DoB, Gender, ZipCode, Position, Salary, Email, Telephone)`

- **Privacy Constraints:** These are described a collection of subsets of columns of a relation which should not be accessible together. The company may have the following privacy constraints defined:
  {Telephone}, {Email}, {Name, Salary}, {Name, Position},{Name, DoB}, {DoB, Gender, ZipCode},{Position, Salary}, {Salary, DoB}

- **Workload:** A workload W is a set of queries that will be executed on a regular basis on the client's data.A possible workload on R could be:

```
SELECT Name
FROM R
WHERE Position = 'Staff';

SELECT *
FROM R
WHERE Salary > 90,000;

SELECT Name, Email, Telephone
FROM R
WHERE Gender = 'F' and ZipCode =
'94305';
```

- **Tuple ID (TID):** Each tuple of the relation is assigned a unique tuple ID. The TID is used to merge data from multiple servers when executing a query on the data. The use of TID will become more explicit in the query plans described next.

- **Partitions:**The server consists of two servers to store the client database.The schema and data is partitioned vertically and stored at the two servers.
  A partition of the schema can be described by three sets R_1 (attributes of R stored on Server 1), R_2 (attributes of R stored on Server 2) and E(set of encrypted attributes stored on both servers). It is important to note that $(R_1 \bigcup R_2 \bigcup E) = R$ and it is not necessarily the case that $R_1 \bigcap R_2 = \phi$. We denote a decomposition of R as D(R).An example decomposition D(R) of R is given here.
  `Partition 1 (`R_1`): (TID, Name, Email, Telephone, Gender, Salary)`
  `Partition 2 (`R_2`): (TID, Position, DoB, Email, Telephone, ZipCode)`
  `Encrypted Attributes (E): Email, Telephone`

- **Query Execution Plans in Distributed Environment:** When data is fragmented across multiple servers, there are two plan types used frequently to execute queries on data stored on these servers.
  Centralized Plans: On execution of a query, data from each server is transmitted to the client and all further processing is done at the client side. In some cases, multiple requests can go the each server but data from one server is never directly sent over to the other servers. Consider the following query for an example of centralized query plan `Query:`

```
SELECT *
FROM R
```

```
WHERE Salary>90,000;
```
In the decomposition D(R), salary is not encrypted. The query is split into the following queries:
```
Query 1:
SELECT TID, Name, Email, Telephone,
Gender, Salary
FROM R1
WHERE Salary > 90,000
Query 2:
SELECT TID,Position,DoB,ZipCode
FROM R2.
```
In the example, the selection on Salary is pushed to the Server 1.At the client side, a join of the two queries over TID is performed to return the results of the initial query.

Semijoin Plans: As an alternative to centralized plans, it is maybe more efficient to consider semijoin plans. Here, TIDs are passed from one server to the other to reduce the amount of traffic flow to the client. For the same example query, the first query using semijoin plans is:
```
Query 1:
SELECT TID, Name, Email, Telephone,
Gender, Salary
FROM R1
WHERE Salary > 90,000
```
The result of Query 1 is returned to the client and the TIDs are passed to Server 2 to get the matching tuples from R2.
```
Query 2:
SELECT TID, Position, DoB,
ZipCode
FROM R2
WHERE TID in <TIDs returned from
Query 1>.
```
The matching tuples from Server 2 are returned back as the results for the query.

- **Encryption Details:** Encryption of columns can either be deterministic or non-deterministic. A deterministic encryption is one which encrypts a column value k to the same value E(k) every time. Thus, it allows equality conditions on encrypted columns to be executed on the server. Our implementation assumes encryption on columns to be deterministic.

- **Column Replication:** When columns of a relation are encrypted, then they can be placed in any of the two servers since they will satisfy all privacy constraints. It is beneficial to store the encrypted columns on both servers to make query processing more efficient. Non encrypted columns can also be duplicated as long as privacy constraints are also satisfied. Replication will result in lesser network traffic most of the time.

- **Cost Overhead:** We model the cost as the number of bytes transmitted on the network assuming that this supersedes the I/O cost on the servers and processing cost on the client. Cost overhead is the parameter used to determine the best possible partitioning of a relation. It measures the number of excess bytes transferred from the server to the client due to the partition.
  Cost Overhead = X - Y,
  where X = Bytes transmitted when executing workload W on a fragmentation F of R at two servers,
  Y = Bytes transmitted when executing workload W on relation R at one server with no fragmentation

The problem can now be defined as follows.
We are given:
(1) A data schema R
(2) A set of privacy constraints P over the columns of the schema R
(3) A workload W defined as a set of queries over R.

We have to come up with the best possible decomposition D(R) of the columns of R into R1,R2 and E such that:
(1) All privacy constraints in P are satisfied. These can either be satisfied by encrypting one or more attributes in the constraint or have at least one column of the constraint at each of the servers. Encrypting columns has its disadvantages as discussed before so we give priority to splitting columns as a way to satisfy privacy constraints.
(2)The cost overhead of D(R) for the workload W should be the minimum possible over all partitions of R which satisfy P. Space is not considered as a constraint and columns of relations are replicated at both servers as long as they satisfy privacy constraints.

Once we come up with the decomposition D(R), given a SQL query posed by the client , we need to partition the query into SQL queries for the appropriate servers. The answers to these queries must then be integrated to return the result of the query at the client.

3 Intractability

A standard framework to capture the costs of different decompositions, for a given workload W, is the notion of the *affinity matrix* [19] M, which we adopt and generalize as follows:

1. The entry M_{ij} represents the performance "cost" of placing the unencrypted attributes i and j in different fragments.

2. The entry M_{ii} represents the "cost" of encrypting attribute i across both fragments.

We assume that the cost of a decomposition may be expressed simply by a linear combination of entries in the affinity matrix. Let $R = \{A_1, A_2, \ldots A_n\}$ represents the original set of n attributes, and consider a decomposition of $\mathcal{D}(R) = \langle R_1, R_2, E\rangle$, where R_1 is at Server 1, R_2 at Server 2 and E the set of encoded attributes. Then, we assume that the cost of this decomposition $C(\mathcal{D})$ is $\sum_{i\in(R_1-E),j\in(R_2-E)} M_{ij}$ + $\sum_{i\in E} M_{ii}$. (For simplicity, we do not consider replicating any unencoded attribute, other than the tupleID, at both servers.

In other words, we add up all matrix entries corresponding to pairs of attributes that are separated by fragmentation, as well as diagonal entries corresponding to encoded attributes, and consider this sum to be the cost of the decomposition.

Given this simple model of the cost of decompositions, we may now define an optimization problem to identify the best decomposition:

Given a set of privacy constraints $\mathcal{P} \subseteq 2^R$ and an affinity matrix M, find a decomposition $\mathcal{D}(R) = \langle R_1, R_2, E\rangle$ such that

(a) $\mathcal{D}$ obeys all privacy constraints in $\mathcal{P}$, and

(c) $\sum_{i,j:i\in(R_1-E),j\in(R_2-E)} M_{ij} + \sum_{i\in E} M_i$ is minimized.

We model the above problem with a graph theoretic abstraction. Each column of the relation is modeled as a vertex of the graph $G(V, E)$, whose edges weights are M_{ij} and vertex weights are M_{ii}. We are also given a collection of subsets $\mathcal{P} \subseteq 2^R$, say $S_1, \ldots, S_t$ which model the privacy constraints. Given this graph $G(V, E)$ with both vertex and edge nonnegative weights, our goal is to partition the vertex set V into three subsets - say RE (the encrypted attributes), R_1 (the attributes at Server 1) and R_2 (the attributes at Server 2). The cost of such a partition is the total vertex weight in E, plus the edge weight of the cut edges from R_1 to R_2. However, the constraint is that none of the subsets $S_1, \ldots, S_t$ can be fully contained inside either R_1 or R_2.

The closely related minimum graph homomorphism problem was studied in [3].

3.1 Minimum Cut when there are Few Sets S_i

There is an algorithm that solves the general problem, but this algorithm is efficient only in special cases, as follows.

Theorem 1 *The general problem can be solved exactly in time polynomial in $\prod_i |Si| = n^{O(t)}$ by a minimum cut algorithm, so the general problem is polynomial if the S_i consist of a constant number of arbitrary sets, a logarithmic number of constant size sets, $O(\log n/\log\log n)$ sets of polylogarithmic size, and $(\log n)^\epsilon$ sets of size $e^{(\log n)^{1-\epsilon}}$ for a constant number of distinct $0 < \epsilon < 1$.*

3.2 Minimum Hitting Set when Solutions do not Use R_2

When edges have infinite weight, no edge may join R_1 and R_2 in a solution.

In the *hitting set problem* we are asked to select a set E of minimum weight that intersects all the sets in a collection of sets S_i.

Theorem 2 *For instances whose edges form a complete graph with edges of infinite weight (so that B, C may not both be used), the problem is equivalent to hitting set, and thus has $\Theta(\log n)$ easiness and hardness of approximation.*

3.3 The case $|S_i| = 2$ and Minimum Edge Deletion Bipartition

When vertices have infinite weight, no vertex may go to E.

In the *minimum edge deletion bipartition problem* we are given a graph and the aim is to select a set of edges of minimum weight to remove so that the resulting subgraph after deletion is bipartite. This problem is constant factor hard to approximate even when the optimum is proportional to the number of edges, as shown by Hastad [14], can be approximated within a factor of $O(\log n)$ as shown by Garg, Vazirani, and Yannakakis [7], and within an improved factor of $O(\sqrt{\log n})$ as shown by Agarwal et al. [1].

The next three results compare the problem having $|S_i| = 2$ to minimum edge deletion bipartition.

Theorem 3 *If all vertex weights are infinite (so that E may not be used), the sets S_i are disjoint and have $|S_i| = 2$, and all edge weights are 1, then the problem encodes the minimum edge deletion bipartition problem and is thus constant factor hard to approximate even when the optimum has value proportional to the number of edges.*

Theorem 4 *If all vertex weights are infinite (so that E may not be used), the sets S_i have $|S_i| = 2$, then the problem may be approximated in polynomial time by a minimum edge deletion bipartition instance giving an $O(\sqrt{\log n})$ approximation.*

Theorem 5 *If all vertex weights are 1, there are no edges, and the sets S_i have $|S_i| = 2$, then the problem encodes minimum edge deletion bipartition and is thus hard to approximate within some constant even for instances that have optimum proportional to the number of vertices.*

We now approximate the general problem with sets S_i having $|S_i| = 2$. The performance is similar to the minimum vertex deletion problem.

Theorem 6 *The general problem with sets S_i having $|S_i| = 2$ can be solved with an approximation factor of $O(\sqrt{n})$ by directed multicut.*

3.4 The case $|S_i| = 3$ and Intractability

The problem with $|S_i| = 3$ becomes much harder to approximate, compared to the $O(\sqrt{n})$ factor for $|S_i| = 2$.

Theorem 7 *If all vertex weights are 1, there are no edges, and the sets S_i have $|S_i| = 3$, then the problem encodes not-all-equal 3-satisfiability and it is thus hard to distinguish instances of zero cost from instances of cost proportional to the number of vertices.*

We examine the tractability when the sets S_i are disjoint.

Theorem 8 *If all vertex weights are infinite (so that E may not be used), the sets S_i are disjoint and have $|S_i| = 3$, and all edge weights are 1, then the problem encodes not-all-equal 3-satisfiability and it is thus hard to distinguish instances of zero cost from instances of cost proportional to the number of edges.*

Theorem 9 *The problem with vertices of infinite weight and edges of weight 1, sets S_i with $|S_i| = 3$ forming a partition with no edges within an S_i, the graph $H_{1,2,3,1',2',3'}$ with no edges joining $S = \{1, 2, 3\}$ and $S' = \{1', 2', 3'\}$ allowed, can be classified as follows:*

(1) If only additional H from K_0 are allowed, the problem is constant factor approximable;

(2) If only additional H from K_0 and K_1 are allowed, the problem is $O(\sqrt{\log n})$ approximable; furthermore as long as some graph from K_1 is allowed, the problem is no easier to approximate than minimum edge deletion bipartition, up to constant factors.

(3) If only additional H from K_0, K_1 and K_2 are allowed, the problem is $O(\log n)$ approximable;

(4) If some additional H from K_3 is allowed it is hard to distinguish instances with cost zero from instances with cost proportional to the number of edges.

We finally note that for dense instances with n vertices, m edges and sets S_i of constant size, we may apply the techniques of Alon et al. [5] to solve the problem within an additive $\epsilon \cdot m$ in time $2^{\tilde{O}(n^2/(\epsilon^2 m))}O(n^2)$ for $m = |E(G)|$.

4 Cost Estimation

Given a schema R, there are different ways to partition the attributes across the servers which satisfy the privacy constraints P. Given also a query workload W, there are some partitions which are more efficient than others in terms of number of bytes transmitted across the network. The problem is to find the best possible partition which minimizes the number of bytes transmitted across the network.

Algorithms with theoretical guarantees are hard to obtain for this problem as explained in Section 3. So, we make use of a greedy hill climbing approach to come up with a decomposition of the input schema. The hill climbing makes use of a cost estimator in its intermediate computations.We describe here how the various components of the system interact with the cost estimator in Figure 2. Given the schema, relation, workload and privacy constraints as input data, an initial partition is determined as explained in Section 5. This partition is fed as input to the hill climbing component. The hill climbing component consists of the translation engine and the cost estimator. The task of the translation engine is to generate queries for the individual partitions given the query over the input schema. The cost estimator determines the cost of the executing the two separate queries over the partitions. The results are returned to the hill climbing component which then decides the next partition to be tried.

In order to evaluate different partitions of the schema, we need to estimate the cost of each possible decomposition with respect to the given workload. The cost overheads can be estimated based on the data collected by standard query optimizers in a database management system.We discuss here the details of a query estimator suitable for the subset of SQL we are interested in for experimental purposes.We describe the cost estimator and translation engine in more detail.

4.1 Query cost estimator

Consider the SQL query

```
SELECT a
FROM T
WHERE F
```

Let
T(R): Number of tuples in R
S(R,a): Size in bytes of attribute a in R

F is a boolean predicate which is given by the grammar in Figure 3. We use $\rho(F)$ to denote the selectivity of the formula F. $\rho(F)$ is computed recursively using the semantics given in Figure 4.

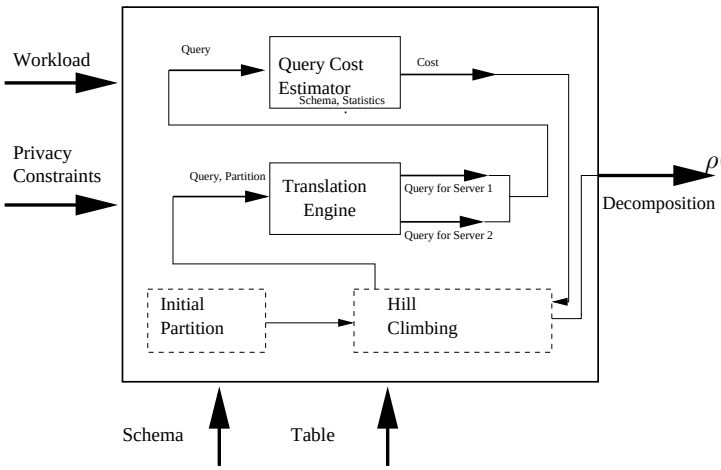

Figure 2. Components of the Partitioning Algorithm

$$
\begin{array}{rrcl}
 & S & ::= & \langle \Sigma, a, c \rangle \\
 & \Sigma & \in & \mathbb{P} \rightarrow \text{Predicate} \\
\text{Predicate} \ni & F & ::= & (F_1 \wedge F_2) \mid (F_1 \vee F_2) \mid a = c \mid \\
 & & & a \leq c \mid c \leq a \mid c_1 \leq a \leq c_2 \mid \\
 & & & a_1 = a_2 \\
 & a & \in & \mathbb{A} = \{ \texttt{T.x}, \texttt{T.y}, ... \} \\
 & c & \in & \mathbb{Z} = \{ ..., -1, 0, 1, ... \}
\end{array}
$$

Figure 3. Syntax of the Boolean Predicate

Here, V(a): Number of distinct values of a in T

The attributes in the SELECT clause of the query decide the size in bytes of each result tuple. Query cost, QC(q) represents the size estimation for query q and SL(q) is the set of attributes in the SELECT clause for q.

$$QC(q) = (\sum_{i \in SL(q)} S(R,i)) * T(R) * \rho(F)$$

4.2 Translation and Execution Engine

The translation engine is the system component which generates queries for the decomposition D(R) of R, given a query on R. The partitioned queries generated by the engine can now be fed to the query estimator discussed in the previous section to obtain cost estimates for each query. The cost estimate for the partition is computed as the sum of the cost estimate of the two queries. The type of plan used is an important factor which decides the form of the resulting queries. For the purposes of this paper, we generate queries for **centralized plans**.

$$
\rho(F) = \begin{cases}
\frac{1}{V(a)} & \text{if } a = c \\
\frac{c - min(a) + 1}{max(a) - min(a) + 1} & \text{if } a \leq c \\
\frac{max(a) - c + 1}{max(a) - min(a) + 1} & \text{if } c \leq a \\
\frac{c_2 - c_1 + 1}{max(a) - min(a) + 1} & \text{if } a_1 \leq c \leq a_2 \\
\frac{1}{max(V(a_1), V(a_2))} & \text{if } a_1 = a_2 \\
1 - (1 - \rho(F_1))(1 - \rho(F_2)) & \text{if } F = F_1 \vee F_2 \\
\rho(F_1) \times \rho(F_2) & \text{if } F = F_1 \wedge F_2
\end{cases}
$$

Figure 4. Semantics of the Boolean Predicate

Replication and encryption pose a lot of issues in query decomposition. For example, if an attribute is available at both servers, one decision to make is which copy must be accessed. Range queries on encrypted attributes will require the entire column to be transmitted to the client for decryption before determining the results of the query. Decisions like which copy to access cannot be determined locally and individually for each condition clause. For example consider the query

```
SELECT Name
FROM R
WHERE Salary=10000 AND Position='Staff'.
```

If Salary is present on both servers and Position only on Server 1, the most efficient way to access the data would be to use the Salary copy on Server 1. Thus, we cannot make a decision as to which copy to use by just examining a single clause like 'Salary =10000'. We propose a technique which computes the where clauses in the decomposed queries in two steps. We define two types of state values, W and S each of which provide information as to which servers to access for the query execution. Detailed description of the state values is provided in the next two subsections. We process the WHERE clause to get W and then process the SELECT clause to get S. In the final step, we use both these values and the corresponding select and condition list to determine the decomposed queries. The details of the algorithm are presented below. We use the schema R and decomposition D(R) defined in section 2.

4.2.1 WHERE clause processing

The WHERE clause of any query is restricted to the clauses that can be generated by the grammar defined in Figure 3. Thus, the basic units of the WHERE clause are conditional clauses where operators could be $>$, $<$ or $=$. Each of these basic units are combined using the AND or the OR operator. The entire clause itself can be effectively represented by a parse tree. Such a parse tree has operators as non-leaf nodes and operands as leaf nodes.

For the query given below,

```
SELECT *
FROM R
WHERE Salary>45000 AND Position='Staff'
```

the parse tree would consist of AND as the root and the two conditions as the left and right subtree of the root.

Bottom Up State Evaluation:

Bottom up evaluation of the parse tree starts at the leaf nodes. Each node transmits to its parent, its state information. The parent operator (always a binary operator) will combine the states of its left and right subtrees to generate a new state for itself.

State Definitions:

Each node in the tree is assigned a state value. Let W be the state value of the root of the parse tree. The semantics of the state value are as follows.

0: condition clause cannot be pushed to either servers.
1: condition clause can be pushed to Server 1.
2: condition clause can be pushed to Server 2.
3: condition clause can be pushed to both servers.
4: condition clause can be pushed to either servers.

As we proceed to determine the state values for all nodes, we need to consider nodes with state value 0 as a special case. Consider the query `select Name from R where Salary > 10,000` and let Salary be deterministically encrypted on the server. The state value for the clause 'Salary $>$ 10,000' is 0 since we cannot push this condition to the server. We need to add the Salary attribute to the list of attributes in the select list since the decomposed query will be `select Name,Salary from R1` assuming Name is stored at Server 1. So, in general, all child attributes for a node with state value 0 are added to the select list of the query.

There are three cases we need to consider for a non-leaf node. **Case 1**: The parent node is one of the operators $>$, $<$, =

For such a parent node, the child nodes are attributes of relations or constant values. If the condition is an attribute-value clause, the state of the parent is the state value of the attribute. The state value of the attribute in turn is determined by the location of that attribute. If the condition is an attribute-attribute clause (a1 = a2 or a1 $<$ a2 etc), the state of the parent is:

0 if the state values of the attributes are (1,2) or if they are on the same server but one of the attributes is encrypted.
1 if the state values of the attributes are (1,1) or (1,4)
2 if the state values of the attributes are (2,2) or (2,4)
4 if the state values of the attributes are (4,4)

Case 2: The parent node is the AND operator

Table 1 represents the state determination of an AND parent node given the state values of its two children. We give an example of how table 1 is used while processing AND clauses .

For the operation $F1 \bigwedge F2$, if F1 and F2 require access to different servers, we push the conditions to the respective servers and perform an intersection of the results at the client side. An example SQL query for the AND case follows.

```
  Query:
SELECT Name
FROM R
WHERE Position='Senior Staff' AND
Salary>'60000'
Query1:
SELECT TID,Name
FROM R1
WHERE Salary>'60000'
Query2:
SELECT TID
FROM R2
WHERE Position='Senior Staff'
```

The client query has two AND predicates, P1 is Position = 'Senior Staff' and P2 is Salary $>$ 60000. The parse tree for this query has two children corresponding to these predicates. The state value for P1 is 1 and P2 is 2. The table shows the state value of the parent to be 3 which implies that the two predicates can be pushed down to the respective servers.

Case 3: The parent node is the OR operator. Table 2 represents the state determination of an OR parent node given the state values of its two children. A similar example for OR clause processing is described here. For the operation $F1 \bigvee F2$, if F1 and F2 require access to different servers, we are forced to bring all columns of interest to the client and process the conditions at the client side.

Consider the following example over the schema R defined in Section 2 which illustrates the behavior of the translation engine.

```
Query:
SELECT Name
FROM R
WHERE Position= 'Senior Staff' OR
Salary > '60000'
Query1:
SELECT TID,Name,Salary
FROM R1
Query2:
SELECT TID,Position
FROM R2
```

Table 1. AND operator state table

	0	1	2	3	4
0	0	3	3	3	3
1	3	1	3	3	1
2	3	3	2	3	2
3	3	3	3	3	3
4	3	1	2	3	4

Table 2. OR operator state table

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	0	0	1
2	0	0	2	0	2
3	0	0	0	0	0
4	0	1	2	0	4

The client query has two OR predicates, P1 is Position = 'Senior Staff' and P2 is Salary $>$ 60000. The parse tree for this query has two children corresponding to these predicates. The state value for P1 is 1 and P2 is 2. The table shows the state value of the parent to be 0 which implies that none of the predicates can be pushed down.

4.2.2 'SELECT' clause processing

The select part of each query can be a set of attributes of the relations in the schema. The select clause processing generates a state S and two sets of attributes A1 and A2. S represents the following cases.
1: Requires access to Server 1 only.
2: Requires access to Server 2 only.
3: Requires access to both servers.
4: Requires access to any of the two servers.

A1 and A2 are the attributes in the select clause that are present on Server 1 and Server 2 respectively. If the attribute is present on both servers, it will be contained in A1 and A2.

4.2.3 Final Query Decomposition step

The final step is to generate the two queries Query 1 (to be sent to Server 1) and Query 2(to be sent to Server 2). We use the state value of the root node obtained from the WHERE clause processing W and the SELECT clause state value S.There are five cases depending on the state value of the root node .
W = 0:
There are no where clauses in Query 1 and Query 2 since none of the conditions can be pushed to the servers. If the select clause state S is 1,2 or 4, we just have a single query (we can process the query by accessing a single server). If S is 3, we add attributes to Query 1 and Query 2 if they are present on Server 1 and server2 respectively.
W = 1 or 2:
For state value 1, Query 2 does not contain any where clauses and for state value 2, Query 1 does not contain any where clauses. If the select clause state S has a value 4, we use the copy of attributes which match with the where clause state value.
W = 3 - Top Down processing of Clauses:
We perform a top-down processing of the tree. We start at the root operator and proceed downwards as long as we encounter state 3. We thus stop when we are sure whether to include the clause as part of Query 1 (state value 1), Query 2(state value 2) , Query 1/Query 2 (state value 4) or not to include it at all(state value 0). The select list attributes can be chosen from any of the servers if they are present on both servers.
W = 4:
The where clauses can be pushed completely to any one of the two servers. We check the select clause state S to determine which of the two queries (Query 1 or Query 2) should contain the where clause. If S is 1,2 or 4, this choice is trivial. If S is 3, we can choose either of the servers to process the where clause.

Let us work with an example to see how all this fits in to solve the problem as a whole. Consider a slightly more complicated client query.

```
SELECT Name, DoB, Salary
FROM R
WHERE (Name ='Tom' AND Position='Staff)
AND (Zipcode = '94305' OR Salary >
60000)
```

Let the predicates in the query be assigned P1 (Name = 'Tom'), P2(Position='Staff'), P3(Zipcode='94305')
and P4(Salary$>$60000) The parse tree and the corresponding state values are shown in Figure 5. Note the state value for OR is 0 since it has 1 and 2 values as its children and we end with a state value W = 3 at the root which is the AND node. The select state is 3 since we need to access both servers to collect attributes specified in the select clause. The select list of attributes has zipcode added to it because of the 0 state so the final select list is Name, Dob, Salary, Zipcode
Since W=3, we perform a top down processing of the parse tree. For P1 and P2, we push them to Server 1 and 2 respectively. For the OR node, we have a 0 state and so we cannot push P3 and P4 to the servers. So we end up with the two partitioned queries Query1 and Query2.

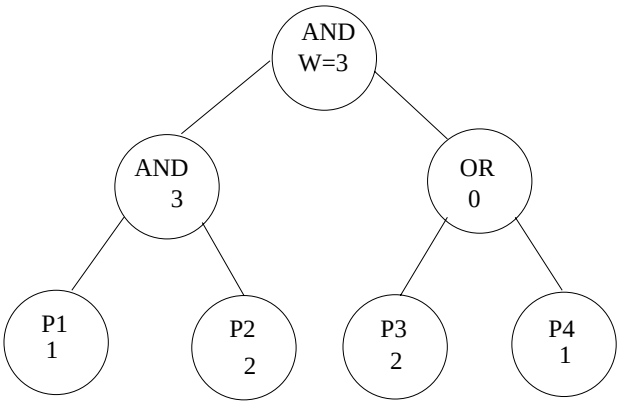

Figure 5. State Computation for the Predicate

```
  Query1:
SELECT TID, name, salary
FROM R1
WHERE Name='Tom'
Query2:
SELECT TID, dob, zipcode
FROM R2
WHERE Position='Staff'
```

The query plan is also shown here in Figure 6 detailing out the steps that need to be performed for executing this query.
In the plan, T,N,S and D stand for TID, Name, Salary and DoB attributes of schema R respectively. At the client side, results of Query1 and Query2 are joined on attribute TID. The predicates P3 and P4 are then applied to the results followed by a projection on the select attributes.

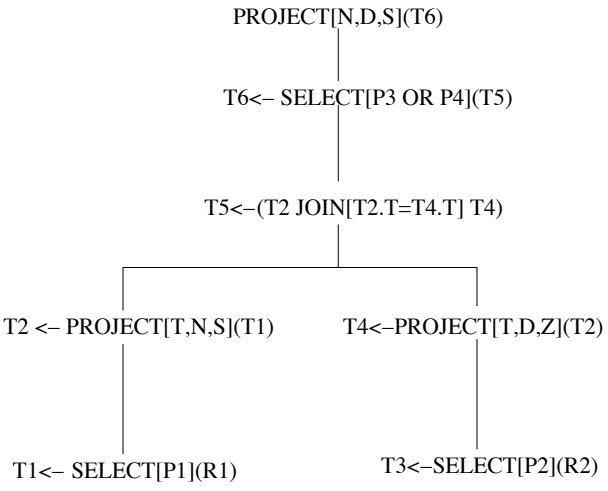

Figure 6. Distributed Query Plan

5 Partitioning Algorithms

As discussed in Section 4, we make use of a greedy hill climbing approach to come up with a decomposition of the input schema. The hill climbing results are not always optimal since they suffer from local minima issues. In order to evaluate the quality of the results of hill climbing we also implement the brute force approach to solve the problem. This approach quickly gets intractable as we increase number of columns of the relation. But for lesser number of columns, this approach can be used to get the optimal decomposition for the given privacy constraints and workload.

5.1 Brute Force Approach

The algorithm is to try out all the possible decompositions of the schema and determine for each the number of bytes transmitted for the given workload. We then select the decomposition which transmits the least number of bytes to be the optimal one. In order to enumerate all the possible decompositions, we try out all possible options for each attribute in turn and generate the decompositions.

Each column of the relation has three options for storage. (1) Store decrypted at Server 1. (2)Store decrypted at Server 2. (3) Store encrypted at both servers. (4) Store decrypted at both servers. The brute force essentially picks the best partition which satisfies all constraints and results in minimum network traffic. Thus, for a relation with n columns there are 4^n possible fragmentations possible and very few of them will satisfy all the privacy constraints.

5.2 Hill Climbing Approach

Hill-climbing is a heuristic in which one searches for an optimum combination of a set of variables by varying each variable one at a time as long as the result increases. The algorithm terminates when no local step decreases the cost. The algorithm converges to a local minima.

An initial fragmentation of the database is considered which satisfies all the privacy constraints.

Initial Guess: The initial guess used as a starting point for the hill climbing algorithm decides how good the final result will be. The initial guess/state is a valid partition of relation R into R1, R2 and E which satisfies all the privacy constraints. A valid initial state is obtained using the weighted set cover.

Algorithm for Weighted Set Cover: - Assign a weight to each attribute based on the number of privacy constraints it occurs in. - Encrypt attributes one at a time starting with the one which has the highest weight till all the privacy constraints are satisfied.

Hill Climbing Step: Then, all single step operations are tried out (1) Decrypting an encrypted column and placing it

at Server 1. (2) Decrypting an encrypted column and placing it at Server 2. (3) Decrypting an encrypted column and placing it at both servers (4)Encrypting an decrypted column and placing it at both servers.

From these steps, the one which satisfies privacy constraints and results in minimum network traffic is considered as the new fragmentation and the process repeats. The iterations are performed as long as we get a decomposition at each step which improves over the existing decomposition using the cost metric discussed before.

6 Experimental Results

6.1 Details of Experimental Setup:

We execute our code on a single relation R in all experiments. The number of attributes in R was varied from 1 to 30. The number of tuples in R was between 1000 and 10,000. (Since we collect and use statistics on the columns of the relation, the experiments/setup would scale easily to larger number of tuples.)

As we vary the number of attributes, we generate privacy constraints over it randomly with the following properties. We generate as many privacy constraints as the number of attributes in the relation. Privacy constraints vary in size from one to the number of attributes in the relation. The attributes that are part of each constraint are selected at random from the available attributes without replacement.
Another important parameter is the workload. We generated 25 workloads containing a fixed number of queries(5 in our case) for different number of attributes of the relations. So, for a relation with fixed number of attributes, we generate about 125 queries (25*5) divided into five workloads. For each query, the parts that were varied were the attribute set in the SELECT clauses and the conditions in the WHERE clause.We selected a subset of SQL which mapped to our grammar defined in Section 4. Each condition clause C was of the form (x OP y) where OP was chosen at random to be $>$,$<$ or $=$. x's were chosen from the columns of R while y was chosen from the domain of x after choosing x. The C's themselves were combined by choosing one of OR,AND. The other parameters that were randomly chosen were the number of condition clauses, the number of attributes in the SELECT clause and the actual attributes in the SELECT clause itself.

6.2 Hill Climbing Vs Brute Force

We conduct experiments to demonstrate how well hill climbing compares with brute force. For each algorithm, we vary the number of attributes N from 1 to 6. We generate ten different workloads for each N and each workload was composed of 5 queries.While we can obtain results for hill climbing for larger N, the brute force approach starts to get intractable. Figure 7 shows the cost difference (in percentage, averaged over the workload) of hill climbing over brute force for varying number of attributes, N. For a given workload, the percentage difference is computed as:

$$\texttt{Percentage Difference} = (C_{hc} - C_{bf})/C_{bf} * 100$$

where C_{hc} = Cost estimate for partition of R returned by Hill Climbing for a given workload W and set of privacy constraints P
and C_{bf} = Cost estimate for the partition with the least cost estimate over all partitions of R for the same workload W and set of privacy constraints P.

Hill climbing starts to move away from the optimal solution with increasing N. However it must be noted that the number of possible partitions considered by brute force increases as 4^N. The difference between hill climbing and brute force also depends on the number of privacy constraints. If there are too many privacy constraints then this restricts the number of viable decompositions to a very small set. Thus, with more number of privacy constraints we get hill climbing results which are more closer to the brute force results.

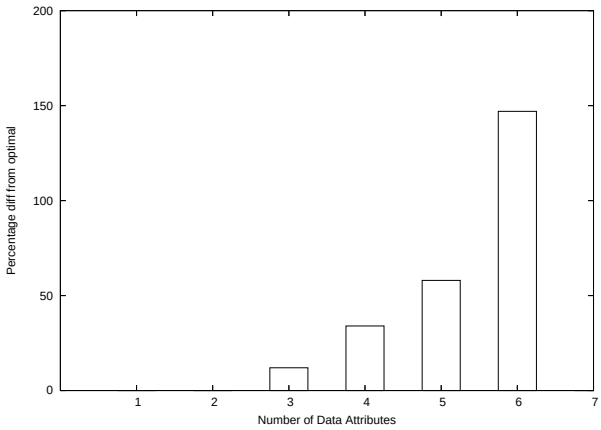

Figure 7. Brute Force vs Hill Climbing

6.3 Hill Climbing Performance

In this set of experiments, we study the behavior of hill climbing in terms of the number of iterations it takes to converge. We also consider to what extent hill climbing has improved the initial partition that we start the algorithm with. We vary the number of attributes N for the relation from 1 to 30 for these experiments.The number of workloads for each N is 10 and similar to the previous experiment, we have 5

queries per workload.
Figure 8 shows the number of workloads for which the hill climbing converged . We note that the number of workloads requiring more than 10 iterations is less than 2 percent of the workloads.

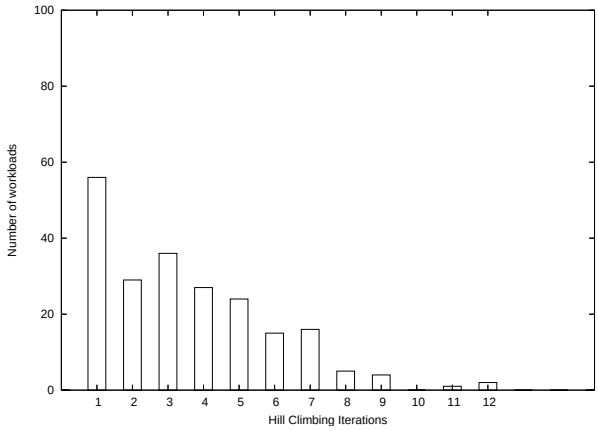

Figure 8. Hill Climbing Iterations

We now show the improvements achieved by the hill climbing over the initial partition. The improvement percentage is defined as:

`Percentage Improvement =` $(C_{fp} - C_{ip})/C_{ip}$ `* 100`

where C_{fp} = Cost estimate for the final partition of R returned by Hill Climbing for a given workload W and set of privacy constraints P
and C_{ip} = Cost estimate of the initial partition given as input to the hill climbing algorithm for the same workload W and set of privacy constraints P.

The graph illustrates that the number of workloads with greater than percent difference from the optimal keeps decreasing with increasing percentage. Despite this fact, more than 50 percent. of the workloads have a percentage improvement of over 25 percent.

6.4 Real World Example

We run experiments for the real world example discussed earlier in the paper in Section 2. The schema R is the same with 8 attributes and 8 privacy constraints. The workloads are generated at random as discussed in the previous subsection. Figure 11 depicts that for about 50 percent of the workloads, the percentage difference of the final result of hill climbing from the initial partition is about 5 percent. Thus, given privacy constraints of the form listed for this example, it is easier to guess a reasonably valid and optimal decomposition for the schema. The fewer number of iterations taken by hill climbing before it terminates is another indicator that optimal or close to optimal decompositions are determined quickly.

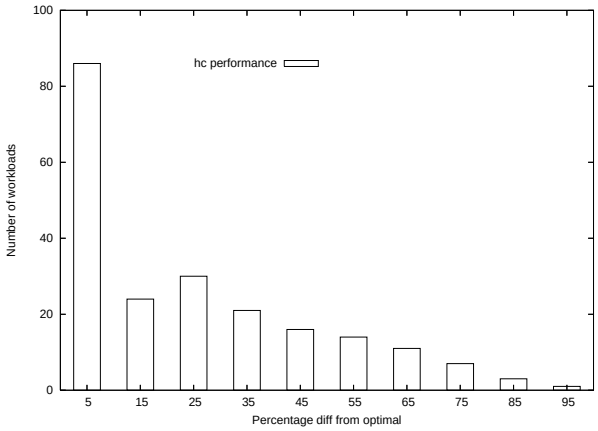

Figure 9. Performance Gain using Hill Climbing

6.5 Varying Distributions of Privacy Constraints Generation Task

We had previously generated a synthetic dataset for the privacy constraints and the workload where all parameters were generated uniformly at random. For this experiment, we use a relation with 8 columns and 1000 tuples. There are 8 privacy constraints generated for the relation. There is a percentage weightage parameter governing the generation of constraints which applies to the first three columns of the relation. For example, if percentage weightage is set to 20, then 60 percent (20*3) of the time, one of the first three columns will be selected as a privacy constraint. So, setting this weightage to around 12 results in a uniform distribution and as it goes above 33, we get a heavily biased set of privacy constraints on these three columns. We generate 30 workloads for each weightage value that we desire to test. Figure 12 shows the average number of iterations as we vary the weightage from 10 to 34 percent. It can be seen that as the bias on the three attributes increases, we get to the final result in fewer number of iterations. This is essentially because most of the privacy constraints are concerned with these three attributes and are independent of the others so we have fewer options to choose for these three attributes.

6.6 Scalability

We discuss here how the algorithms discussed are scalable as we increase the number of attributes and tuples in the relation. One way to handle huge amounts of data is to use statistics. For the given schema, we compute and store

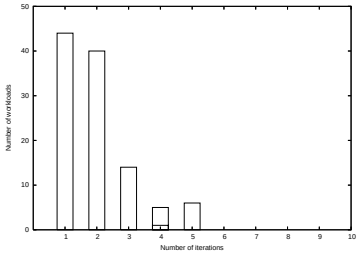

Figure 10. Real World Example - Hill Climbing Iterations

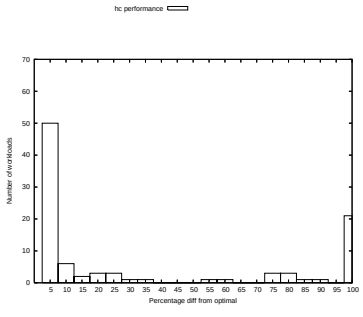

Figure 11. Real World Example - Performance Gain using Hill Climbing

the statistics of the relation and the columns so no further queries to the database are necessary. Thus, the partitioning algorithms scale for arbitrary large number of tuples in the relation.

The complexity of hill climbing is related to the number of attributes and each iteration takes linear time in the number of attributes. As seen in our experiments, the number of iterations depends on the set of privacy constraints P and the workload W.For a randomly generated set of privacy constraints and workload, we find that the iteration count does not show signs of an increasing trend. This allows us to increase the number of attributes with the hill climbing algorithm scaling linearly.

7 Related Work

There is a wide consensus that privacy is a corporate responsibility [16]. In order to help and ensure corporations fulfil this responsibility, governments all over the world have passed multiple privacy acts and laws, for example, Gramm-Leach-Bliley (GLB)Act [8], Sarbanes-Oxley (SOX) Act [21], Health Insurance Portability and Accountability Act (HIPAA) [15] are some such well known U.S. privacy acts. In many use cases complying with these laws require an organization to encrypt the data in case it is hosted by an external service provider.

As discussed in the introduction, the outsourcing of data management has motivated the model where a DBMS provides reliable storage and efficient query execution, while not knowing the contents of the database [13]. Schemes proposed so far for this model encrypt data on the client side and then store the encrypted database on the server side [12, 11, 4]. However, in order to achieve efficient query processing, all the above schemes only provide very weak notions of data privacy. In fact a server that is secure under formal cryptographic notions can be proved to be hopelessly inefficient for data processing [17]. Our architecture of using multiple servers helps to achieve both efficiency and provable privacy together. Closest to our work is that of fragmenting data for privacy proposed in [18]. However they do not consider this as an optimization problem where the aim is to reduce network traffic.

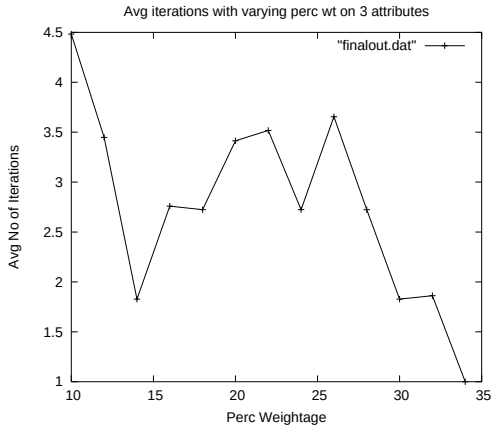

Figure 12. Average Iterations with varying privacy constraints distribution

7.1 Distributed Databases - Semijoin plans

The algorithms discussed in this paper are closely related to query processing in vertically fragmented databases. Semijoin plans are considered to be very effective for partitioned databases. We have an example of semijoin plans in Section 2. We have concentrated on centralized plans in this paper and we do not consider semijoin plans for the following reasons.

1) In a distributed database, all software components at each site is under the control of a single software vendor. This vendor can decide the kind of interfaces, capabilities that can/should be supported by each of the sites. The servers in our case are equivalent to the remote sites in distributed databases. Each maybe an independent service provider and the only assumption in the provider supports SQL interface to the database. Thus, all semijoin plans need to be specified as SQL queries and data cannnot be impliclity transported.

2) Consider a simple query of the form: `select TID,name from R where DoB=1982 and Salary=40000` Let us assume that DoB and Salary are in different partitions. If we use a semijoin plan, we come up with a plan where we send all tuple IDs from Server 1 with DoB=1982 to Server 2. The query result for Server 2 will be the final result of the query. So the service provider for Server 2 can specifically determine which tuples matched the final query. This information in some sense is a leak in the privacy since all the selected tuples have a special value for DoB which is a sensitive attribute when combined with Salary.

References

[1] A. Agarwal, M. Charikar, K. Makarychev, and Y. Makarychev. $o(\sqrt{\log n})$ approximation algorithms for min uncut, min 2cnf deletion, and directed cut problems. In *Proc. 37th Ann. ACM STOC*, pages 573–581, 2005.

[2] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep a secret: A distributed architecture for secure database services. In *Conference on Innovative Data Systems Research*, 2005.

[3] G. Aggarwal, T. Feder, R. Motwani, and A. Zhu. Channel assignment in wireless networks and classification of minimum graph homomorphism. In *ECCC TR06-040*, 2006.

[4] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order-preserving encryption for numeric data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2004.

[5] N. Alon, W. F. de la Vega, R. Kannan, and M. Karpinski. Random sampling and approximation of max-csp problems. In *J. Comput. Syst. Sci.*, pages 212–243, 2003.

[6] Amazon. Amazon elastic compute cloud. Available from URL: http://www.amazon.com/b/ref=sc_fe_l_2/?node=201590011&no=3435

[7] N. Garg, V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. In *SIAM J. Comp.*, pages 235–251, 1996.

[8] GLB. Gramm-Leach-Bliley Act. Available from URL: http://www.ftc.gov/privacy/privacyinitiatives/glbact.html.

[9] Google. Google apps for your domain. Available from URL:http://www.google.com/a/.

[10] A. Gupta. Improved results for directed multicut. In *Proc. 14th Ann. ACM-SIAM SODA*, pages 454–455, 2003.

[11] S. M. H. Hacigumus, B. Iyer. Efficient execution of aggregation queries over encrypted relational databases. In *Proc. DASFAA*, 2004.

[12] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.

[13] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing database as a service. In *Proceedings of the International Conference on Data Engineering*, 2002.

[14] J. Hastad. Some optimal inapproximability results. In *Proc. 29th Annual ACM Symp. on Theory of Computing*, pages 1–10, 1997.

[15] HIPAA. Health Information Portability and Accountability Act. Available from URL: http://www.hhs.gov/ocr/hipaa/.

[16] IBM. Privacy is good for business. Available from URL: http://www-306.ibm.com/innovation/us/customerloyalty/harriet_pearson_interview.shtml.

[17] M. Kantarcioglu and C. Clifton. Security issues in querying encrypted data. Technical Report TR-04-013, Purdue University, 2004.

[18] A. Motro and F. Parisi-Presicce. Blind custodians: A database service architecture that supports privacy without encryption. In *International Federation for Information Processing*, 2005.

[19] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition, 1999.

[20] Salesforce. Salesforce on-demand customer relationship management. Available from URL:http://www.salesforce.com/.

[21] SOX. Sarbanes-Oxley Act. Available from URL: http://www.sec.gov/about/laws/soa2002.pdf.

8 Appendix

8.1 Minimum Cut when there are Few Sets S_i

`Theorem` 1. The general problem can be solved exactly in time polynomial in $\prod_i |Si| = n^{O(t)}$ by a minimum cut algorithm, so the general problem is polynomial if the S_i conist of a constant number of arbitrary sets, a logarithmic number of constant size sets, $O(\log n/\log\log n)$ sets of polylogarithmic size, and $(\log n)^\epsilon$ sets of size $e^{(\log n)^{1-\epsilon}}$ for a constant number of distinct $0 < \epsilon < 1$.

Proof. One may try in all possible ways to select for each S_i either one element to go to E or two elements to go to B, C respectively. Merge together the identified elements of each of A, B, C next, remove the identified elements in E, and now we are looking for a min cut between the identified B, C elements, where the cut is given by vertices that go to E and edges that join B, C. In fact we may turn the edges into vertices by putting a middle vertex of the appropriate weight on each edge, so we are looking for a min-vertex cut, which is polynomial. Thus the complexity is polynomial in $\prod_i |S_i| = n^O(t)$ because of the initial number of possible choices. □

8.2 Minimum Hitting Set when Solutions do not Use R_2

`Theorem` 2 For instances whose edges form a complete graph with edges of infinite weight (so that B, C may not both be used), the problem is equivalent to hitting set, and thus has $\Theta(\log n)$ easiness and hardness of approximation.

Proof. We may not cross the cut B, C as this would give infinite cost. We may thus assume that only A, B will be used. Each set S_i must then have at least one element in E, so a solution is valid only if E hits all the sets S_i, and the cost is the sum of the vertex weights in the hitting set E. □

8.3 The case $|S_i| = 2$ and Minimum Edge Deletion Bipartition

`Theorem` 3. If all vertex weights are infinite (so that E may not be used), the sets S_i are disjoint and have $|S_i| = 2$, and all edge weights are 1, then the problem encodes the minimum edge deletion bipartition problem and is thus constant factor hard to approximate even when the optimum has value proportional to the number of edges.

Proof. Encode each vertex v of G as $S_v = \{a_v, b_v\}$, and encode each edge vw of G as the two edges joining S_v, S_w given by $a_v b_w$, $b_v a_w$. The side of the bipartition for v depends on whether a_v or b_v goes to R_1, and an edge vw is removed if v, w go to the same side, in which case we pay for both $a_v b_w$, $b_v a_w$ across the cut B, C. Thus the problem is equivalent to the minimum edge deletion bipartition. □

`Theorem` 4. If all vertex weights are infinite (so that E may not be used), the sets S_i have $|S_i| = 2$, then the problem may be approximated in polynomial time by

a minimum edge deletion bipartition instance giving an $O(\sqrt{\log n})$ approximation.

Proof. If two sets S_i share an element, say $S_i = \{a, b\}$ and $S_j = \{a, c\}$, then we may merge b and c. We may thus assume the S_i are disjoint. Now represent $S_v = \{a_v, b_v\}$ by a vertex v, and if S_v, S_w are joined by edges we must pay at least one for these edges, unless these edges are (1) contained in $a_v b_w, b_v a_w$ or (2) contained in $a_v a_w, b_v b_w$. In case (1) we join vw by an edge, and in case (2) we introduce a new vertex u and form a path vuw of length 2. This encodes the problem as a minimum edge deletion bipartition problem up to a costant factor, so an $O(\sqrt{\log n})$ approximation exists. The problem with edge weights is similarly solved by subtracting weights joining S_i and S_j until we fall in cases (1) or (2) above.

□

`Theorem` 5. If all vertex weights are 1,there are no edges, and the sets S_i have $|S_i| = 2$, then the problem encodes minimum edge deletion bipartition and is thus hard to approximate within some constant even for instances that have optimum proportional to the number of vertices.

Proof. If we consider the sets S_i as edges, this is the problem of removing the least number of vertices to make the graph bipartite. We know hardness for removing edges to make the graph bipartite. To translate to vertices, separate each vertex of degree d into d vertices, one for each adjacency, and connect these d vertices with a constant degree expander graph, where each edge of the expander graph is replaced with constant number of parallel paths of length two. Thus if less than half the vertices of the same expander graph get removed, this corresponds to removing a proportional number of edges. □

`Theorem` 6. The general problem with sets S_i having $|S_i| = 2$ can be solved with an approximation factor of $O(\sqrt{n})$ by directed multicut.

Proof. Represent each vertex v of weight x by four vertices a_v, b_v, c_v, d_v joined by arcs $a_v b_v, c_v d_v$ of capacity x. Represent each $S_i = \{u, v\}$ by arcs $b_u c_v, d_u a_v, b_v c_u, d_v a_u$ of infinite capacity. Represent each edge uv of weight y by arcs $b_u a_v, d_u c_v, b_v a_u, d_v c_u$ of capacity y. Finally look for a multicut that separates the sources a_v from the corresponding sinks d_v. Removing an arc for a vertex x corresponds to assigning x to E, and after removing arcs corresponding to uv, the vertices form by reachability two components R_1 for arcs $a_v b_v$ and R_2 for arcs $c_v d_v$. The multicommodity flow result of Gupta [10] for directed multicut gives the $O(\sqrt{n})$ bound. □

8.4 The case $|S_i| = 3$ and Intractability

`Theorem` 7. If all vertex weights are 1, there are no edges, and the sets S_i have $|S_i| = 3$, then the problem encodes not-all-equal 3-satisfiability and it is thus hard to distinguish intances of zero cost from instances of cost proportional to the number of vertices.

Proof. If there are are no edges and the sets S_i have size $|S_i| = 3$, then the problem encoded is not-all-equal 3-satisfiability by corrresponding sets R_1 and R_2 to values 0 and 1 respectively. Even satisfiable instances of not-all-equal 3-satisfiability have a constant factor hardness on the number of variables participating in unsatisfied clauses by a solution. We conclude that a constant fraction of such variables must be assigned to E even if a zero cost solution exists. The hardness of approximation of non-all-equal satisfiability for number of variables instead of clauses is obtained by making multiple copies of the same variable for multiple clauses, and joining these with a constant degree expander graph. Each edge xy of the expander graph represents a path of length two xzy, where xz and zy represent $x \neq z$ and $z \neq y$ over $\{0, 1\}$ respectively. We represent $x \neq y$ with clauses $\{x, t, y\}, \{x, u, y\}, \{x, v, y\}, \{t, u, v\}$. The result thus folows from the result for not-all-equal satisfiability of Hastad [14]. □

`Theorem` 8 If all vertex weights are infinite (so that E may not be used), the sets S_i are disjoint and have $|S_i| = 3$, and all edge weights are 1, then the problem encodes not-all-equal 3-satisfiability and it is thus hard to distinguish instances of zero cost from instances of cost proportional to the number of edges.

Proof. A clause of not-all-equal satisfiability may be viewed as a set $S_i = \{x_i, y_i, z_i\}$. We assume these S_i are disjoint and join copies of variables in different clauses by a clique. If the not-all-equal 3-satisfiability problem has a solution, a solution of zero cost exists for our problem. The number of clauses satisfied in a not-all-equal satisfiability problem is constant factor hard to approximate even on instances that are satisfiable. Therefore in a solution to our problem a constant fraction of the sides chosen for the elements of the S_i would have to be changed between R_1 and R_2 to obtain a consistent solution to not-all-equal 3-satisfiability that fails a constant fraction of the clauses. If we replace each clique by a constant degree expander graph, then each of the elements of S_i that would be changed between R_1 and R_2 pays a constant, as at most half of the elements of the expander graph for a clique are changed. Thus we pay cost proportional to the number of edges when the optimal cost is zero by hardness of approximation of not-all-equal 3-satisfiability shown by Hastad [14]. □

Theorem 9 The problem with vertices of infinite weight and edges of weight 1, sets S_i with $|S_i| = 3$ forming a partition with no edges within an S_i, the graph $H_{1,2,3,1',2',3'}$ with no edges joining $S = \{1,2,3\}$ and $S' = \{1',2',3'\}$ allowed, can be classified as follows:

(1) If only additional H from K_0 are allowed, the problem is constant factor approximable;

(2) If only additional H from K_0 and K_1 are allowed, the problem is $O(\sqrt{\log n})$ approximable; furthermore as long as some graph from K_1 is allowed, the problem is no easier to approximate than minimum edge deletion bipartition, up to constant factors.

(3) If only additional H from K_0, K_1 and K_2 are allowed, the problem is $O(\log n)$ approximable;

(4) If some additional H from K_3 is allowed it is hard to distinguish instances with cost zero from instances with cost proportional to the number of edges.

Proof. We prove (4). The case of $H_{11',2,2',3,3'}$ was proven in the preceeding theorem. The case of $H_{11',22',3,3'}$ simulates $H_{11',2,2',3,3'}$ by considering $H_{11'',22'',3,3''}, H_{1''\hat{1},3''\hat{3},2'',\hat{2}}, H_{\hat{1}1',\hat{2}2',\hat{3},3'}$. The case of $H_{11'2',2,3,3'}$ simulates $H_{11',2,2',3,3'}$ by considering $H_{11''2'',2,3,3''}, H_{1''2''1',2',3',3''}$. This proves (4).

We next prove (1). Any occurrences of $H_{1231',2',3'}, H_{1231'2',3'}, H_{1231'2'3'}$ must pay in a solution, so we may remove these and pay cost proportional to the number of these. The graphs $H_{121'2',3,3'}, H_{121'2',33'}$ are equivalent up to constant factors, so we consider just $H_{121'2',33'}$. If 12 are combined in $H_{121'2',33'}$ and 13 are combined in $H_{131''3'',22''}$, then at least one of these two graphs must pay, so for each set $S = \{1,2,3\}$ we may consider the number a_{12} of graphs combining 12, the number a_{13} of graphs combining 13, and the number a_{23} of graphs combining 23, and remove the least two of a_{12}, a_{13}, a_{23} number of graphs (say remove the graphs for combinations 13 and 23 and keep the graphs for combinations 12). This incurs another constant factor of the optimum. Finally every S has only one combination 12, so we may assign 12 to R_1 and 3 to R_2 at zero cost. This proves (1).

We next prove (2). Define $\hat{H}_{11',22',3,3'}$ by $H_{11'',22''3'',3}, H_{1'1'',2'2''3'',3'}$. Thus $\hat{H}_{11',22',3,3'}$ is $H_{11',22',3,3'}$ plus the condition that $11'$ and $22'$ go to R_1 and R_2 respectively or to R_2 and R_1 respectively. We may remove occurrences of the first three graphs H in K_0 by paying cost proportional to the number of such H as before. The last two H in K_0 and the two H in K_1 can be simulated by $\hat{H} = \hat{H}_{11',22',3,3'}$, as they are superpositions of several copies of $\hat{H}$ under various permutations of the elements of S and S', and superpositions may be avoided by concatenating two copies of $\hat{H}$ to obtain $\hat{H}$ again. We may thus suppose that $\hat{H}$ is the only graph that occurs. Suppose the role of 3 in $\hat{H}$ for $S = \{1,2,3\}$ is played in different groups by $1,2,3$, so that we have $\hat{H}_{11',22',3,3'}$, $\hat{H}_{11'',33'',2,2''}$, $\hat{H}_{2\hat{2},3\hat{3},1,\hat{1}}$. Then one of these three must pay, so we may remove for S the one group that occurs in the least number of such graphs (say keep the first to with the role of 3 played by 2 or 3 and remove the ones where the role of 3 is played by 1). The cost payed is proportional to the number of such graphs removed, incurring a constant factor approximation. We may finally assume that only $\hat{H}_{12',21',3,3'}$, $\hat{H}_{13',31',2,2'}$ occur. If $\hat{H}_{11',22',3,3'}$ occurs, it can be simulated as $\hat{H}_{12'',21'',3,3'}$, $\hat{H}_{1'2''',2'1'',3,3'}$. Thus we may say that 1 crosses in such occurrences with either 2 or 3, and so the instance can be solved at zero cost if and only if the graph whose vertices are the S_i and the edges are the $\hat{H}_{12',21',3,3'}$ joining them is bipartite. The problem thus reduces to the minimum edge deletion bipartition problem and is solvable in time $O(\sqrt{\log n})$. It can be shown that as long as some graph in K_1 is allowed, the problem is no easier than minimum edge deletion bipartition, up to constant factors. This proves (2).

We finally prove (3). Suppose first only $H_{11',22',33'}$ occurs. This graph permutes 123 into $1'2'3'$ in some way. We may compose such permutations and come back to $S = \{1,2,3\}$. If 123 comes back as 231, then the instance has no solution of zero cost. If this never happens, say 123 only comes back as 213, then we may map $1,2$ to R_1 and 3 to R_2, obtaining a solution of zero cost. We may thus represent each such S by six vertices corresponding to the six permutations $123, 132, 213, 231, 312, 321$, and match the six permutations for $S = \{1,2,3\}$ to the six permutations for $S' = \{1',2',3'\}$. On this graph with six vertices for each S, we may look for a multicut separating each pair $123, 231$ for each S. This can be done by the algorithm of Garg, Vazirani, and Yannakakis [7] with an $O(\log n)$ approximation. This proves the case of $H_{11',22',33'}$ alone.

We complete the proof of (3). The first three sets H of K_0 pay as from before and are removed at a constant factor approximation. The last two sets H of K_0 and the two sets H of K_1 can be simulated by $\hat{H}_{11',22',3,3'}$ defined as from before. The part of the problem involving only $H_{11',22',33'}$ can be solved with an $O(\log n)$ approximation, by removing the corresponding multicut. Now if a set S is connected directly to d other S_i, then create d copies of S, one for each S_i, and join the copies of S with a constant degree expander graph involving edges between this copies having $H_{11',22',33'}$. Finally for each occurrence of $\hat{H}_{11',22',3,3'}$ at $S = \{1,2,3\}$ (there is now at most one such occurrence at S) ask for a multicut separating $1,2$ for such S, as $1,2$ must go to B, C respectively or C, B respectively. This is again done with an $O(\log n)$ approximation by the algorithm of Garg, Vazirani, and Yannakakis [7]. This completes the proof of (3). □