Distributed SQL Queries with BubbleStorm

Christof Leng and Wesley W. Terpstra

Databases and Distributed Systems, Technische Universität Darmstadt, Germany
{cleng,terpstra}@dvs.tu-darmstadt.de

Abstract. Current peer-to-peer (p2p) systems place the burden of application-level query execution on the application developer. Not only do application developers lack the expertise to implement good distributed algorithms, but this approach also limits the ability of overlay architects to apply future optimizations. The analogous problem for data management was solved by the introduction of SQL, a high-level query language for application development and amenable to optimization.

This paper attempts to bridge the gap between current access-oriented p2p systems and relational database management systems (DBMS). We outline how to implement every relational operator needed for SQL queries in the BubbleStorm peer-to-peer overlay. The components of BubbleStorm map surprisingly well to components in a traditional DBMS.

1 Introduction

Due to its advantages SQL became the most widely used query language for structured data. It combines a powerful set of query operations with a relatively clean abstraction of system internals. The abstraction benefits both the user and the DBMS developer. A user can easily learn and use SQL without understanding how the DBMS executes his queries. The DBMS developer on the other hand has the freedom to optimize the execution of the query in many different ways. This combination leads to an easy to learn yet performant query language. Beyond that SQL offers flexibility for both sides. The user can introduce new queries or change existing ones at any time. The DBMS developer can integrate new optimizations without breaking existing applications. The advantages of SQL have made it almost ubiquitous in computing. Many standalone applications like Firefox use SQL internally to manage their data and countless hobby programmers use MySQL and the like for their web projects.

In contrast to the SQL success story, things in peer-to-peer look quite bleak. The best-known interface for peer-to-peer search overlays is *key-based routing* (KBR) [5]. It provides an abstraction that works more or less with all *distributed hash tables* (DHT), but is not much more than a hash table interface. This limits users to simple key-value lookups or forces them to build tailor-made algorithms for their more sophisticated problems on top of that primitive. The execution plan of such a query is thus moved from the realm of the system architect to the responsibility of the user. Obviously this contradicts the idea of an abstraction like SQL. Now the user needs to have the expertise to implement search algorithms and the system architect can not optimize the execution easily.

K. Sachs, I. Petrov, and P. Guerrero (Eds.): Buchmann Festschrift, LNCS 6462, pp. 230–241, 2010.

With key-value lookup as the only means of access, DHTs resemble indexes in traditional database systems. Indexes are an important tool for database performance tuning, but with ever increasing hard disk throughput and nearly constant random access times, their relevance is decreasing. Instead, scanning the table directly is often more efficient than using the index. The rise of solid state disks with their super fast random access might change the rules here, but there won't be such a thing for the Internet. The latency of Internet connections is dictated by the physical distance between the communication partners and the speed of light. On the other hand according to Gilder's Law [4] the bandwidth doubles or even triples each year. Or as David Clark once put it "There is an old network saying: bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed - you can't bribe God."

Therefore, a more powerful abstraction for peer-to-peer search overlays should be based on a technology that is able to benefit from continuing bandwidth growth. Such an abstraction might be based on the established SQL standard. This would enable developers to build upon their database programming experience for peer-to-peer application development resulting in an easier learning curve. In this paper we present how to implement distributed SQL queries with the BubbleStorm [12] peer-to-peer search overlay. Its rendezvous approach makes BubbleStorm a perfect fit for this kind of abstraction because it resembles a table scan in a traditional database system.

We cover all major aspects of SQL queries like selection, projection, and aggregation. A focus of this paper is the discussion of join algorithms applicable in BubbleStorm. Furthermore query execution planning for the peer-to-peer environment is discussed. The data definition language (DDL) for schema modifications and the data manipulation language (DML) for inserts and updates are beyond the scope of this paper.

2 Related Work

Distributed search is probably the most prominent research topic in peer-to-peer networking. Most publications focus on algorithms for more efficient or more powerful search and relatively few propose abstractions or query languages for those algorithms. Of those, key-based routing [5] is the most well-established approach. Unfortunately, it is so low-level that it should be used to build application-level query languages rather than direct application development. Also, it focuses on DHTs which are limited to indexing functions and cannot be scanned efficiently.

A few projects have considered SQL as a query language for their peer-to-peer system. All of them have in common that they keep local databases that are shared with other users instead of a global database that is distributed over the network. Thus, when a user goes offline, he takes his data with him. Furthermore, a user with a large amount of data might become overloaded with requests and will be unable to answer all of them reliably.

PeerDB [9] was one of the earliest attempts at accessing distributed data with SQL queries. In the tradition of multi-database systems they assume no general schema and apply a schema matching algorithm that relies on human interaction. The underlying naïve flooding-based overlay is clearly not scalable.

Minerva [3] is a peer-to-peer database based on DHTs. The DHT is used to index the local databases of the participating peers. To execute a query, one looks up the most promising peers for the relevant query terms in the DHT and then queries them directly. The distributed index might become prohibitively expensive if the number of terms and users grows. Like PeerDB, queries that combine data from multiple nodes like joins are not considered.

Astrolabe [14] is a peer-to-peer information dissemination system that allows queries to be formulated in SQL. As an information dissemination system it does not support storing data in the network but queries the current state of online nodes. Astrolabe is organized hierarchically and thus relatively static. Sacrificing one of the major advantages of SQL, it only allows predefined queries because it pre-aggregates information in its hierarchy.

3 BubbleStorm Overview

BubbleStorm [12] is a rendezvous-based peer-to-peer search overlay. A rendezvous search system is a distributed system that ensures that a query meets all data that is available in the system. Meeting means that the query is executed on at least one node that has a copy of the data item in its local dataset. This is typically achieved by replicating both data and queries onto $O(\sqrt{n})$ nodes. The benefit of a rendezvous system is that any selection operator that can be executed locally can be executed in the distributed search overlay.

Designed for highly dynamic, failure-prone and open-membership scenarios BubbleStorm does not guarantee an unobtainable 100% search success, but instead gives a tunable probabilistic success guarantee. Replicating query and data to $\sqrt{\lambda n}$ nodes guarantees a search success probability of $p = 1 - e^{-\lambda}$. Thus, the application developer can pick the right traffic vs. recall tradeoff for his application by setting λ.

This probabilistic approach is implemented by a random graph topology which gives BubbleStorm extreme resilience against node failures [12]. Every node can pick its own degree and will keep this number of neighbours. Since the load of a node in BubbleStorm scales linearly with its degree, this ensures load balance in heterogeneous environments.

The actual replication of data and queries in the overlay is implemented with a binary-tree-based algorithm called bubblecast. Being a tree, bubblecast reaches x nodes within $\log x$ hops. The intersection of a given query and data bubble is called the rendezvous (Figure 1). The nodes in the rendezvous (there could be several) can test this data item for the searcher's query.

To compute the global parameters for bubblecast and topology maintenance BubbleStorm monitors a number of system statistics like the network size. To do this it uses a gossip-based measurement protocol [13] that computes system-wide sums, maxima, and averages and provides the results to all nodes in the system.

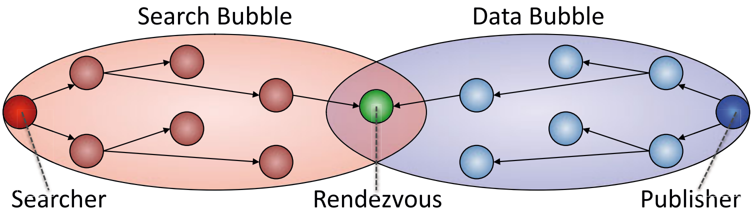
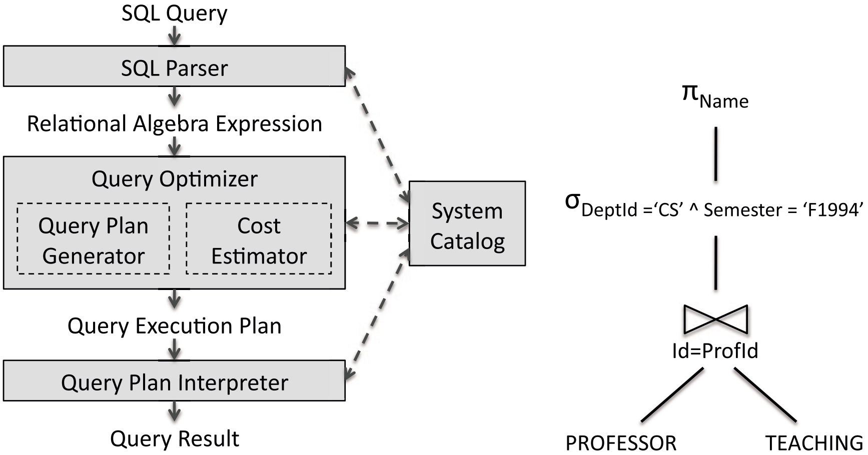


Fig. 1. Intersection of two bubbles in BubbleStorm

4 SQL Query Optimization

Our proposal for executing SQL queries closely follows the approach used in modern DBMS systems. To understand it, a short review of standard SQL query processing is in order. For a more complete understanding, the interested reader is referred to [7].

Architecturally, the system goes through the steps shown in Figure 2(a). First, the SQL query is parsed into an expression involving relational operators. Then, various result-preserving rearrangements of these operators, called query plans, are considered by the optimizer. The plan generator creates the rearrangements and the cost estimator determines how expensive they would be. Finally, the chosen query plan is fed into the interpreter which executes the query.



(a) Query processing
(b) An example query tree

Fig. 2. DBMS query execution (taken from [7])

The SQL Parser needed for a BubbleStorm DBMS is unchanged from a traditional DBMS system. It produces relational expressions like those in Figure 2(b).

The plan generator also remains more-or-less unchanged, considering only left-deep plans using a dynamic programming algorithm [7]. Where the BubbleStorm DBMS differs is the plan interpreter, cost estimator, and system catalog. Section 5 will detail how BubbleStorm can execute each relational operator and derive a cost estimator for each technique. Section 6 details how to implement a DBMS catalog in a peer-to-peer setting.

5 Plan Execution in BubbleStorm

Before discussing how we choose an optimized query plan, we first turn our attention to how we execute a given plan in BubbleStorm. Armed with this understanding, we will be able to estimate the cost of a plan and thus choose an acceptable strategy.

To execute a query plan in a peer-to-peer network, we pay several different costs. First, there is the time required for participating peers to locally execute the query, which we ask a traditional DBMS to estimate. As every operation includes this classically estimated cost, we elide them from discussion to keep things simple. A real implementation should include the local execution cost in its estimates during query optimization. We will henceforth focus on the bandwidth costs.

Depending on the execution plan, the bandwidth costs may be paid by the executing peer (we call this local bandwidth cost) or by intermediate peers in the network (we call this network bandwidth cost). Query plans can trade between a very selective, but large query (imposing a large network bandwidth cost) or a smaller, simple query (requiring more local bandwidth to filter the results). The correct trade-off between these local/network costs depends on the application and resources of the executing peer. In our cost analysis, we assign a simple weighting: $\mathscr{L}$ for local bandwidth cost and $\mathscr{N}$ for network bandwidth cost. When discussing bandwidth costs, message headers play an important constant-wise role. We will use h to designate their length, typically on the order of 100 bytes in a BubbleStorm implementation.

5.1 Selection and Projection

As discussed in Section 3, BubbleStorm provides a rendezvous search interface. This lends itself naturally to the execution of selection and projection on a base table. Since both selection and projection may be executed simultaneously, we consider a combined select/project operator.

To execute the operator, we define two intersecting bubble types. One bubble type consists of the rows (or tuples) in the base table. The other bubble type contains a particular select/project operator. To be concrete, each bubble of the first type contains a tuple insertion statement like, "`INSERT INTO tableA VALUES (4, 5, 6);`". These insert statements are individually replicated onto several of the participating peers. Each bubble of the second type contains a select statement like, "`SELECT project-clause FROM tableA WHERE select-clause;`", which is executed by several participating peers.

Each peer locally stores the tuples it receives in a traditional database. Upon receipt of a projection/selection operator, it uses this database to execute the operator and then forwards the results to the originator of the query. BubbleStorm ensures that every tuple and every operator meet on some peer with a user-specified probability, $p = 1 - e^{-\lambda}$. Therefore, the query originator receives p of the requested tuples. Unfortunately, each matching tuple might be received multiple times (on average, λ times).

The cost to execute this operator has several parts. First there is the cost of shipping the query to the n executing peers. If the selection/projection-operator has length q, then the originator as the root of the bubblecast tree pays $2(h+q)$ traffic to send the query to two peers, but the network pays $2(h+q)m$ by replicating the query to $m \approx \sqrt{\lambda n}$ peers for the rest of the tree. This includes the cost of sending and receiving the query.

Finally, there is the cost to download the result-set. Suppose the result table R has r entries and a total size of $|R|$ bytes. The header cost depends on the number of responding peers. Recall that λ BubbleStorm peers respond for each tuple, but only one of them will transfer the payload. The expected number of responders is asymptotically $1 - e^{-\lambda r/m}$ percent of the m peers, making the bandwidth cost to both the originator and the network $hm(1 - e^{-\lambda r/m}) + |R|$ traffic. The total cost is therefore,

$$(\mathscr{L} + \mathscr{N})[2(h+q) + hm(1 - e^{-\lambda r/m}) + |R|] + \mathscr{N} 2m(h+q) \tag{1}$$

In some cases, we may have an index available, allowing us to find tuples without a full scan of the table. BubbleStorm has its own indexing mechanism costing e hops on average (DHTs can do the same in $\log n$ hops). If an index is available, the download cost and query forwarding costs remains unchanged. However, $m \approx e$ instead of $\approx \sqrt{\lambda n}$, a potentially significant savings whenever an index is available.

5.2 Post Processing

After receiving matching tuples, the query originator might need to aggregate and sort/filter the result. The user may have specified an SQL aggregation operation like, "`SELECT owner, SUM(balance) FROM account GROUP BY owner HAVING SUM(balance) < 0;`". He may also have requested the output in sorted order, possibly with distinct results. We perform all of this post processing locally on the originator where standard database techniques apply.

Performing aggregation is relatively inexpensive so long as the result set fits in main memory. Form a hash-table using the `GROUP BY` expression as the key. Then as results arrive, aggregate them in-place. All of the standard SQL aggregates can be implemented this way; `AVG` (average) can be implemented using `SUM` and `COUNT`. In our scenario, with a slow Internet connection and desktop-class peers, this work can be pipelined with receiving the incoming tuples. If the aggregated result does not fit in main memory, there are standard algorithms to perform aggregation very efficiently. Their cost can be estimated from the estimated size of the result set.

Similarly, sorting the result set can be achieved by storing the incoming tuples in a balanced search tree as they arrive. If the group-by columns appear only as a prefix of the order-by columns, sorting can be combined with aggregation by replacing the hash-table with a balanced search tree. Otherwise sorting must occur after aggregation. If main memory is insufficient and/or sorting cannot be combined with aggregation, the cost of an external sort can be estimated from the size of the result set.

One interesting enhancement is to move the aggregation into the network. A query like "`SELECT COUNT(*) FROM account;`" can clearly be further optimized. Some of the previous work on peer-to-peer search result retrieval [1] builds a tree, where the aggregation proceeds up the tree. This increases the total traffic, since the leaf peers still send the same results and now intermediate peers must also transmit. However, it does lighten the burden of the originator and parallelizes the load. As a more fruitful approach, one could move the aggregation operator into the leaf peers themselves. Then, each peer reports only its locally aggregated result.

Unfortunately, the result sets from each peer in BubbleStorm are not disjoint. This leads to double-counting. Leaf peer aggregation can be applied to rendezvous systems which guarantee exactly one copy of each result, like [11,10,2]. Eliminating double-counting in a more failure-tolerant scenario like BubbleStorm would be an interesting direction for future work.

5.3 (Block) Nested Loop Join

In a traditional DBMS, a nested loop join is used when one table (S) is small. For each tuple in the smaller table, the larger table (L) is scanned for matching tuples. The block nested optimization simply loads a block of S at a time instead of a tuple. During the scan of L, tuples are matched against the loaded block S.

In some sense, a nested loop join writes a subquery like,

$$\texttt{SELECT * from } L \texttt{ WHERE joinColumn=}x\texttt{;} \tag{2}$$

for each tuple $x \in S$. The block nested variant reads like,

$$\texttt{SELECT * from } L \texttt{ WHERE joinColumn in } X\texttt{;} \tag{3}$$

for each block $X \subseteq S$. We use this analogy to create a BubbleStorm equivalent.

To execute a block nested loop join in BubbleStorm, first query the network to load the result table S as in Section 5.2. Now there are two options. If an index is available for the join columns in L, we can use query 2 to lookup the join results one at a time. Alternatively, divide S up into 'blocks' that fit inside a query bubble. For each block B, run the query 3 with $X = B$. This select operation can be executed as described in Section 5.1 using rendezvous techniques.

Now we compute the costs. Let $\pi(S)$ be the projection of the columns necessary to perform the join and s the total number of tuples in S. When there is an index, we replace m with e and q with $|\pi(S)|$ in Equation 1 and run it s times,

$$(\mathscr{L} + \mathscr{N})[2sh + 2|\pi(S)| + she + |R|] + \mathscr{N}2e(sh + |\pi(S)|) \tag{4}$$

Similarly, performing a block-nested loop join costs,

$$(\mathscr{L} + \mathscr{N})[2h + 2|\pi(S)| + hm(1 - e^{-\lambda r/m}) + |R|] + \mathscr{N}2m(h + |\pi(S)|) \quad (5)$$

Whenever an index is available and $s < m \approx \sqrt{\lambda n}$, using an index-nested loop join is beneficial. However, if $s > m$ things start to swing the other way. In some sense, this transition captures the selectivity of the query, analogous to a traditional DBMS. As S grows, you need to execute more and more subqueries and batching them together in a scan become cheaper.

5.4 Sort-Merge and Hash Join

In a traditional DBMS, nested loop joins are not ideal for similarly sized, large tables. This is also true for BubbleStorm due to the factors he for index-nexted loop joins and m for block-nested loop joins. For the case where both tables are large, DBMSs use either a sort-merge or hash join.

Ignoring the details, both sort-merge and hash join walk both tables once in their entirety. Sort-merge works well when the tables are already sorted and hash join partitions the tables while it walks them. When an index is present, a nested loop join may be faster since each tuple in the outer table O can find matching inner tuples without walking the entire inner table I. However, there is a cost to this lookup, and once a significant portion of the inner table is retrieved, it becomes faster to simply walk the inner table in its entirety.

The situation in BubbleStorm is more-or-less analogous. The rendezvous system provides an index for any select criteria. Nevertheless, it has a cost m to use per outer tuple. Once $|O|m > |I|$, it becomes cheaper to walk the entire I.

In the setting of BubbleStorm, walking both tables in their entirety simply means to retrieve both tables (after applying any selection and projection operators). This can be done as in Section 5.1. Since a block nested loop join must already retrieve the smaller table, it is quite intuitive that retrieving both is a good plan when they have similar size. After both tables have been retrieved, they can be joined locally using a traditional hash join. Hash join is preferred since the tuples do not arrive in sorted order, and partitioning can be easily pipelined with reception.

6 Query Plan Generation and the Catalog

Armed with equations to estimate the cost of given query plan, we can now pick a good one. The considerations are identical to those in a traditional DBMS; we choose the best left-deep tree using a dynamic programming algorithm [7]. Just as in a traditional DBMS, we push the projection and selection operations as far toward the leaves in the query plan as we can. If a table has an index on the join columns, we also consider a plan which does not push selection to the base table (which would prevent use of the index). The query plan generation for a

BubbleStorm search system is thus completely standard. The only difference is the particular costs for the operators and how we obtain the size estimates for resulting relations.

To compute the cost of our operators, we need to know the size of the result. A traditional DBMS estimates these sizes using the database catalog. The catalog contains, at the least, the database schema, the number of rows in each table, the average bit length of each attribute, the number of distinct values for each attribute, and the maximum/minimum for each attribute. This information can be used to estimate result sizes. For example, if a table `Teachers` is joined by `EmployeeID` with another table which has 20 distinct `EmployeeIDs`, the catalog can be used to estimate the number of matching tuples in `Teachers`. Take the ratio of the distinct `EmployeeIDs` to the total number of tuples in the `Teachers` table (in our example this ratio is probably 1). Now multiply this by 20 to determine that we expect 20 tuples in the result. Sum the average column sizes for the projected columns and multiply by 20 to find the resulting table size.

In a peer-to-peer database like BubbleStorm, the schema is a global piece of metadata. In order to facilitate future development, new, signed versions of the schema can be flooded to all participating peers. Care must be taken between subsequent versions of the software to ensure that the schema is kept backwards compatible with the program logic. Nevertheless, this part of the catalog is relatively straight-forward to access.

BubbleStorm already includes a mechanism for computing sums, averages, minimums and maximums. This measurement protocol can be used to find the average size of each column, the total number of tuples in the table, and the minimum/maximum for each attribute. The global schema tells peers which attributes and tables to gossip about using the measurement protocol.

The only piece of information that is hard to come by in a peer-to-peer setting is the number of distinct values in an attribute. The heart of the problem is that we have no locality and there are duplicate entries. A peer with one value has no way of knowing how many other peers have the same value. However, there is a way to calculate the number of distinct values using statistics.

It is a well-known property of k independent exponential random variables X_i with rate γ that their minimum $Y := \min_i X_i$ is also exponentially distributed with rate $k\gamma$. This can be exploited, as in [8], to calculate sums by finding k. While this technique is inferior to the distributed sum algorithm used in BubbleStorm [6,13], it can be tweaked to calculate the number of distinct objects (a variation we have not seen published yet).

To count the distinct values in a table's attribute, hash each value to obtain a seed for a random number generator. Use this seed to compute an exponential random variable with $\gamma = 1$. Find the minimum of the exponential random variables computed locally. Use BubbleStorm's built-in measurement protocol to find the global minimum Y. Take $1/Y \approx k$ as the number of distinct objects.

This algorithm works because two copies of the same object result in the same seed. Therefore, there are only as many dice rolled as there are distinct objects. Unfortunately, an exponential random variable has standard deviation equal to its mean. However, by averaging several of these minimums, the estimate may be improved. If we average j minimums, the standard deviation falls to $k/\sqrt{j}$. In a practical system where measurements are continuously generated, we can take an exponentially moving average of the minimums. If the weight given to the newest estimate is $1/16$, then the standard deviation is better than ± 25%, good enough for our cost estimates.

7 Materialized Views

When a join is executed many times, one way to improve performance is to cache/store the joined table. While traditional DBMS systems do not create materialized views on their own, given a view created by the administrator, most query optimizers will recognize when the view can be used. In this way, materialized views can be used to improve system performance.

Updating a materialized view is quite complicated. When the base tables change, a subquery must be triggered which updates the view. These subqueries may themselves be using auxiliary tables to speedup execution. We believe that most of these trigger-based approaches could be applied one-to-one in a BubbleStorm DBMS.

However, peer-to-peer rendezvous gives us another possibility. We can create a materialized cross-product of two tables at a cost which, while still expensive, is much cheaper than in a traditional DBMS. The core idea is to form a three-party rendezvous instead of the more common two-party scenario. Local to each peer the cross-product need not be materialized; it can execute the join operator directly or perhaps materialize the join instead.

If a query is interested in a join of tables A and B, then a three-way rendezvous will ensure that every pair of tuples $(a, b) \in A \times B$ will rendezvous with the query. The idea is that if there are enough copies of a and b, then many peers will have both of them. When the query is executed, some peer who has both will also receive the query. This allows that peer to execute the join operation completely locally and still produce the resulting tuple.

While we never actually store the cross-product on disk, the increased replication of tables A and B is obviously expensive. Normally, if the query had q replicas and table A a replicas, these must roughly obey $qa = \lambda n$ which leads to $q, a \in O(\sqrt{\lambda n})$. With the materialized cross-product, the relationship is $qab = \lambda n^2$, so $q, a, b \in O(\sqrt[3]{\lambda n^2})$. If one of the relations is small or infrequently changed, replicating it to every node may be less expensive than executing the join repeatedly. Conversely, if the join query is rarely executed, the cost to nearly flood it might be acceptable.

We leave as future work the problem of gauging when/if a three-way rendezvous is cheaper than a trigger-based update for materialized views.

8 Conclusion

In this paper we described how to build a complete SQL query processor for the BubbleStorm peer-to-peer network. The main thrust of our work is how to execute selection, aggregation, index and block nested loop joins, and hash joins. For each operator we provided a cost estimator compatible with a traditional DBMS query optimizer. The cost estimator relies on the system catalog. We designed a distributed version of the system catalog which we show can be done with just BubbleStorm's gossip protocol.

We believe traditional DBMS architecture remains a good fit in the peer-to-peer environment. In particular BubbleStorm provides everything needed to arrive at a simple and natural design. Table scans map to bubblecast, indexes map to key-value lookups, and the system catalog maps to the gossip protocol. Overall it is surprisingly easy to execute SQL queries with BubbleStorm.

References

1. Balke, W.-T., Nejdl, W., Siberski, W., Thaden, U.: Progressive distributed top-k retrieval in peer-to-peer networks. In: Proceedings of ICDE 2005, Washington, DC, USA, 2005, pp. 174–185. IEEE Computer Society Press, Los Alamitos (2005)
2. Barroso, L.A., Dean, J., Hölzle, U.: Web search for a planet: The google cluster architecture. IEEE Micro. 23(2), 22–28 (2003)
3. Bender, M., Michel, S., Triantafillou, P., Weikum, G., Zimmer, C.: Minerva: collaborative p2p search. In: Proceedings of VLDB 2005, Trondheim, Norway, pp. 1263–1266. VLDB Endowment (2005)
4. Coffman, K.G., Odlyzko, A.M.: Internet growth: is there a "moore's law" for data traffic? In: Handbook of massive data sets, pp. 47–93. Kluwer Academic Publishers, Norwell (2002)
5. Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., Stoica, I.: Towards a common api for structured peer-to-peer overlays. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, Springer, Heidelberg (2003)
6. Kempe, D., Dobra, A., Gehrke, J.: Gossip-Based Computation of Aggregate Information. In: Proceedings of FOCS 2003, Washington, DC, USA, p. 482. IEEE Computer Society Press, Los Alamitos (2003)
7. Kifer, M., Bernstein, A., Lewis, P.M.: Database Systems: An Application Oriented Approach, Compete Version. Addison-Wesley, Reading (2006)
8. Mosk-Aoyama, D., Shah, D.: Computing separable functions via gossip. In: PODC 2006: Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, pp. 113–122. ACM, New York (2006)
9. Ng, W.S., Ooi, B.C., Tan, K.-L., Zhou, A.: PeerDB: A P2P-based system for distributed data sharing. In: Proceedings of ICDE 2003 (2003)
10. Raiciu, C., Huici, F., Handley, M., Rosenblum, D.S.: Roar: increasing the flexibility and performance of distributed search. In: Proceedings of SIGCOMM 2009, pp. 291–302. ACM, New York (2009)
11. Terpstra, W.W., Behnel, S., Fiege, L., Kangasharju, J., Buchmann, A.: Bit Zipper Rendezvous—Optimal Data Placement for General P2P Queries. In: Lindner, W., Mesiti, M., Türker, C., Tzitzikas, Y., Vakali, A.I. (eds.) EDBT 2004. LNCS, vol. 3268, pp. 466–475. Springer, Heidelberg (2004) (received best paper award)

12. Terpstra, W.W., Kangasharju, J., Leng, C., Buchmann, A.P.: Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. In: Proceedings of SIGCOMM 2007, pp. 49–60. ACM Press, New York (2007)
13. Terpstra, W.W., Leng, C., Buchmann, A.P.: Brief announcement: Practical summation via gossip. In: Twenty-Sixth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2007), pp. 390–391. ACM Press, New York (August 2007)
14. Van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems (TOCS) 21(2), 164–206 (2003)