Parallelizing Skyline Queries for Scalable Distribution

Ping Wu, Caijie Zhang, Ying Feng, Ben Y. Zhao, Divyakant Agrawal and Amr El Abbadi

University of California at Santa Barbara
{pingwu,caijie,yingf,ravenben,agrawal,amr}@cs.ucsb.edu

Abstract. Skyline queries help users make intelligent decisions over complex data, where different and often conflicting criteria are considered. Current skyline computation methods are restricted to centralized query processors, limiting scalability and imposing a single point of failure. In this paper, we address the problem of parallelizing skyline query execution over a large number of machines by leveraging content-based data partitioning. We present a novel distributed algorithm that discovers skyline points progressively. We propose two mechanisms, recursive region partitioning and dynamic region encoding, to enforce a partial order on query propagation in order to pipeline query execution. Our analysis shows that DSL is optimal in terms of the total number of local query invocations across all machines. In addition, simulations and measurements of a deployed system show that our system load balances communication and processing costs across cluster machines, providing *incremental scalability* and significant performance improvement over alternative distribution mechanisms.

1 Introduction

Today's computing infrastructure makes a large amount of information available to consumers, creating an information overload that threatens to overwhelm Internet users. Individuals are often confronted with conflicting goals while making decisions based on extremely large and complex data sets. Users often want to optimize their decision-making and selection criteria across multiple attributes. For example, a user browsing through a real-estate database for houses may want to minimize the price and maximize the quality of neighborhood schools. Given such a multi-preference criteria, the system should be able to identify all potentially "interesting" data records. Skyline queries provide a viable solution by finding data records not "dominated" by other records in the system, where data record x dominates y if x is no worse than y in any dimension of interest, and better in at least one dimension. Records or objects on the skyline are "the best" under some monotonic preference functions[1].

A more general variant is the *constrained skyline query* [19], where users want to find skyline points within a subset of records that satisfies multiple "hard" constraints. For example, a user may only be interested in car records within the price range of \$10,000 to \$15,000 and mileage between 50K and 100K miles. The discussion hereafter focuses on this generalized form of the skyline query.

[1] Without loss of generality, we assume in this paper that users prefer the minimum value on all interested dimensions.

Until recently, Skyline query processing and other online analytical processing (OLAP) applications have been limited to large centralized servers. As a platform, these servers are expensive, hard to upgrade, and provide a central point of failure. Previous research has shown common-off-the-shelf (COTS) cluster-based computing to be an effective alternative to high-end servers [4], a fact confirmed by benchmarks [5] and deployment in large query systems such as Google [7]. This approach is also being adopted in commercial database systems such as Oracle 10g. In addition, skyline queries are especially useful in the context of Web information services, where user preferences help form "structured" queries across a large number of distributed data sources [3, 2]. For these web services, a scalable distributed approach can significantly reduce processing time, and eliminate high query load during peak hours.

Our paper is the first to address the problem of distributing progressive skyline queries on a share-nothing architecture through content-based data partitioning and exploiting partial orders between data partitions. This paper makes four key contributions. First, we present a recursive region partitioning algorithm and a dynamic region encoding method. These algorithms enforce the skyline partial order so that the system pipelines participating machines during query execution and minimizes inter-machine communication. As a query propagates, our system prunes data regions and corresponding machines for efficiency, and progressively generates partial results for the user. In addition, we propose a "random sampling" based approach to perform fine-grain load balancing in DSL. Next, we perform analysis to show that our approach is optimal in minimizing number of local query invocations across all machines. Finally, we describe the cluster deployment of a full implementation on top of the CAN [21] content distribution network, and present thorough evaluations of its bandwidth, scalability, load balancing and response time characteristics under varying system conditions. Results show DSL clearly outperforms alternative distribution mechanisms.

The rest of the paper is organized as follows: Section 2 describes our design goals as well as two simple algorithms for distributed skyline calculation. We present our core algorithm (DSL) in Section 3. In Section 4, we address the query load-balancing problem in DSL. We then evaluate our system via simulation and empirical measurements in Section 5. Finally, we present related work in Section 6 and conclude in Section 7.

2 Design Goals and Proposals

In this section, we describe our design goals for parallel/distributed skyline query processing algorithms. We then present two simple solutions and discuss their limitations.

2.1 Goals

In addition to basic requirements for skyline processing, we describe three design goals for a distributed skyline algorithm: *progressiveness, scalability,* and *flexibility*.

Progressiveness. Similar to the requirements for centralized solutions [17], a distributed algorithm should be able to progressively produce the result points to the user: *i.e.*, the system should return partial results immediately without scanning the entire data set. Progressiveness in a distributed setting further requires that results be returned without involving all the nodes in the system. This eliminates the need for a centralized point for result aggregation.

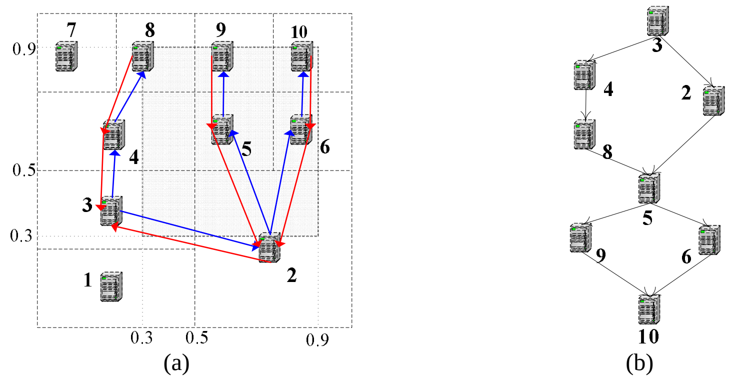


Fig. 1: (a) CAN multicast-based Method with in-network pruning. (b) Observation: partial order between nodes.

Scalability. Incremental scalability is the primary goal for our system. In order to scale to a large number of participant machines, we require that internode communication be minimized, and processing load should be spread evenly across all nodes. It should also be easy to add more nodes into the system to handle increased data volume and/or heavier query load.

Flexibility. Our goal for flexibility has two components. First, the system should support *constrained skyline queries*, and find skyline records in arbitrarily specified query ranges. Second, the distributed algorithm should not impose any restrictions on the local implementation on each machine, thus allowing easy incorporation of "state of the art" centralized skyline solutions.

2.2 Simple Solutions

In this section, we discuss two simple approaches towards distributing data and query processing across multiple machines. We analyze both proposals according to our stated goals.

Naive partitioning. One simple approach is to partition data records randomly across all machines, and to contact all nodes to process each query. Each node calculates a result set from local data, and all result sets are merged at a centralized node. To reduce congestion, we can organize the nodes into a multi-level hierarchy where intermediate nodes aggregate result sets from children nodes. We call this approach the *naive method*.

While easy to implement, this approach has several drawbacks. First, each query must be processed by *all* nodes even if the query range is very small, resulting in significant unnecessary computation. Second, most data points transmitted across the network are not in the final skyline, resulting in significant waste in bandwidth. Finally, this method is not progressive, since the final result set cannot be reported until all the nodes have finished their local computations. Note that using locally progressive algorithms does not produce globally progressive results.

CAN Multicast. An improved algorithm utilizes the notion of content-based data partitioning. Specifically, we normalize the entire data space and directly map it to a virtual coordinate space. Each participating machine is also mapped into the same coordinate space and is responsible for a specific portion of that space. Then every machine stores all the data points that fall into its space. During query processing, a multicast tree is built to connect together all nodes overlapping with the query range, with the root at the node that hosts the bottom-left point of the query range [2]. The query propagates down the tree, nodes perform local computation, and result sets are aggregated up back to the root. Ineligible data points are discarded along the path to preserve bandwidth. Figure 1(a) illustrates how the tree is dynamically built at query time. Node 3 hosts the bottom-left point of the query range ((0.3, 0.3),(0.9, 0.9)), and acts as the multicast tree root. In this paper, we implement the content-based data partitioning scheme by leveraging the existing code base of the CAN [21] content distribution network. Therefore we call this approach the *CAN-multicast* method. While the following discussion is based on the CAN overlay network, our solutions do not rely on the specific features of the CAN network such as decentralized routing and are thus applicable to general cases of content-based data partition as well.

The CAN-multicast method explicitly places data so that constrained skyline queries only access the nodes that host the data within the query range. This prunes a significant portion of unnecessary data processing, especially for constrained skyline queries within a small range. However, its nodes within the query box still behave the same as those in the naive method. Thus it shares the bandwidth and non-progressiveness drawbacks. In fact, both methods can be seen as direct adaptation of the original centralized "Divide and Conquer" method [8].

3 Progressive Distributed Skylines

In this section, we begin by making observations from exploring the simple methods described in the last section. Based on these observations, we propose our progressive skyline query processing algorithm (DSL) and show the analytical results regarding its behavior.

3.1 Observations

Our progressive algorithm derives from several observations. Using the CAN multicast method, no result can be reported until results from *all* nodes in the query range are considered. We note that this strategy can be optimized by leveraging content-based data placement. Skyline results from certain nodes are guaranteed to be part of the final skyline, and can be reported immediately. For example, in Figure 1(a), no data points from other nodes can dominate those from node 3, and node 3 can reports its local results immediately. Meanwhile, node 8's calculations must wait for results from 3 and 4, since its data points can be dominated by those two nodes.

On the other hand, data points in nodes 4 and 2 are mutually independent from a skyline perspective; that is, no points from node 2 can dominate points in node 4 and vice versa. Therefore, their calculations can proceed in parallel. We summarize this observation as: any node in the virtual CAN space can decide whether its local skyline points are in the final result set or not by only consulting a *subset* of all the other nodes within the query range (***Observation 1***).

[2] The choice of the root will not impact the final result set as long as all the nodes in the query range are covered by the tree

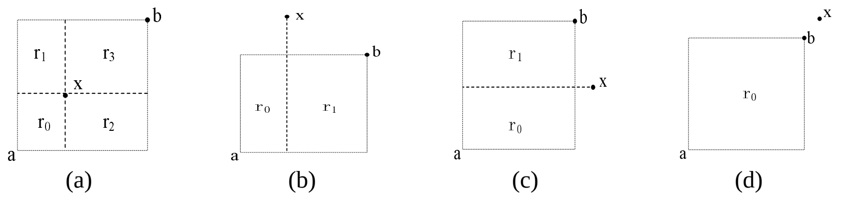


Fig. 2: Region partitions on 2-d CAN space.

Based on the *Observation 1*, we visualize the computational dependency between nodes in Figure 1(b). Each edge in the graph captures the precedence relationship between CAN nodes. During query propagation, skyline points must be evaluated at all "upstream" nodes before "downstream" nodes can proceed. Based on this, we also observe that based on skyline values from upstream nodes, some nodes within the query region do not need to be contacted (***Observation 2***). For example, in Figure 1(a), any skyline point from node 3 will dominate all points in nodes 5, 6, 9 and 10. Therefore, any skyline results from node 3 means computation at those nodes can be skipped.

3.2 Partial Orders over Data Partitions

We now formalize the notion of partial order over data partitions. According to CAN terminology, we call each data partition a *zone* in the CAN virtual space. Let Q_{ab} be a d-dimensional query region in CAN space; $a(a_1, a_2, ..., a_d)$, $b(b_1, b_2, ..., b_d)$ be the bottom-left and top-right points, respectively. The *master node* of Q_{ab}, denoted as $M(Q_{ab})$, is the CAN node whose zone contains the point a (*e.g.* Node 3 is the master node in Figure 1(a)).

Let point $x(x_1, x_2, ..., x_d)$ be the top-right point of $M(Q_{ab})$'s CAN zone (*e.g.* point (0.5,0.5) in Figure 1(a)). $M(Q_{ab})$ partitions the query region Q_{ab} as follows: for each dimension $i (1 \leq i \leq d)$, if $x_i < b_i$, $M(Q_{ab})$ partitions Q_{ab} into two halves on dimension i: namely the upper interval $[x_i, b_i]$ and the lower interval $[a_i, x_i]$; if $x_i \geq b_i$, the partition will not occur on this dimension since $M(Q_{ab})$ "covers" Q_{ab} on dimension i. Thus, $M(Q_{ab})$ divides the query space Q_{ab} into at most 2^d subregions (e.g., the query region in Figure 1(a) is partitioned into 4 subregions by node 3). We denote all the subregions resulting from the partition as the *region set* $RS(Q_{ab})$ and $|RS(Q_{ab})| \leq 2^d$.

Example 1. Figure 2 shows all four possibilities for region partitioning on a 2-d CAN space. a, b determine the query box Q_{ab} and x represents the top-right point of $M(Q_{ab})$'s zone. In (a), $RS(Q_{ab})$ contains 4 subregions (denoted as r_0, r_1, r_2, r_3) since x falls inside the query box and both dimensions are split. In (b) and (c), only one dimension is divided since x is greater than b in at least one dimension. Therefore, $RS(Q_{ab})$ contains 2 subregions (denoted as r_0, r_1) in both cases. Finally, in (d), the zone of $M(Q_{ab})$ covers the entire query space (on both dimensions), and no partitioning occurs. $\square$

Given a query region Q_{ab} and its master node's zone, the region partitioning process as well as its resulting region set $RS(Q_{ab})$ can be uniquely determined. It is important to note that the region partitioning process is *dynamically* determined, depending on 1) query region Q_{ab} of the current skyline query; 2) the CAN zone of the node $M(Q_{ab})$ containing the virtual coordinate a. Furthermore, since there does not exist a "global" oracle in a distributed setting and each node only sees its own zone, this process is executed at the master node $M(Q_{ab})$. Next we define a partial order relation on $RS(Q_{ab})$.

Definition 1. *(Skyline Dependent, $\ll$): Relation "Skyline Dependent, $\ll$" is a relation over Region Set $RS(Q_{ab})$: region r_i is "Skyline Dependent" on region r_j, i.f.f. $\exists p(p_1, p_2, ..., p_d) \in r_i, \exists q(q_1, q_2, ..., q_d) \in r_j$, s.t. $\forall k, 1 \leq k \leq d, q_k \leq p_k$, i.e., q "dominates" p.*

Example 2. In Figure 2(a), there are four subregions resulting from the partition of the query region Q_{ab}. Specifically, $RS(Q_{ab}) = \{r_0, r_1, r_2, r_3\}$. And according to Definition , $r_1 \ll r_0$, $r_2 \ll r_0$, $r_3 \ll r_0$, $r_3 \ll r_1$ and $r_3 \ll r_2$.

Theorem 1. *"Skyline Dependent, $\ll$" is a reflexive, asymmetric, and transitive relation over $RS(Q_{ab})$, and thus it defines a partial order over the region set $RS(Q_{ab})$.*

Proof: It is straightforward to show the reflectivity and transitivity of "Skyline Dependent". Asymmetry can be derived by the fact that all the subregions resulting from the region partitioning process are convex polygons. □

Intuitively, for each incoming query, if we can control the system computation flow to strictly satisfy the above partial order, then we can produce skyline results progressively. Hence nodes in a region would not be queried until they see the results from *all* "Skyline Dependent" regions. The reason for this is that with the aid of the partial order between regions, the local skyline on each participant node is only affected by the data in its "Skyline Dependent" regions, i.e. each region is able to determine its final result based *only* on the data from its "Skyline Dependent" regions and its own data records. This exactly captures our previous two observations.

3.3 Dynamic Region Partitioning and Encoding

We still face two remaining challenges. The first challenge involves generalizing the above approach to the case where subregions are distributed over multiple CAN zones. We call this the *Resolution Mismatch Problem*. We address this challenge with a *Recursive Region Partitioning* technique. Specifically, for a query range Q_{ab}, for each subregions in $RS(Q_{ab})$ resulting from a region partitioning based on master node $M(Q_{ab})$, the same region partitioning process is carried out recursively. Since after one region partitioning, at least the bottom-left subregion r_0 is entirely covered by the zone of $M(Q_{ab})$, we can resolve one part of the region Q_{ab} at each step. Consequently, this recursive process will terminate when the entire query region is partitioned and matches the underlying CAN zones. Figure 3(a) shows that for the query range ((0.3,0.3),(0.9,0.9)), in total, region partitioning process is invoked three times on node 3, 2, and 6 sequentially until each of the resulting subregions is covered exactly by one CAN zone.

The second challenge that naturally arises is that the query range for a constrained skyline query is only given at *query time*. The recursive region partitioning and the partial order information are also computed at query time, since they are completely dependent on the query range. In order to enforce the partial order during the query propagation in a distributed setting, the master nodes in the subregions should know the predecessors they need to hear from before their own regions are activated, as well as their successive regions that it should trigger upon its own completion. Below, we

Algorithm 1 Successor Calculation

```
1: $Q_{ab}$: the “parent” region of $r_{cd}$;
2: $r_{cd}$: a region $\in RS(Q_{ab})$;
3: $ID(r_{cd})$: the code of $r_{cd}$;
4: $succ(r_{cd})$: successors of region $r_{cd}$;
5: $succ(r_{cd}) \longleftarrow \emptyset$; //Initialization
6: foreach $i$,s.t. $ID(r_{cd})[i]$== ’0’
7: begin
8: $oneSucessor.code[i] \longleftarrow$ ’1’; // flip one ’0’ bit to ’1’
9: $oneSucessor.region[i] \longleftarrow$($d[i]$,$b[i]$ ); // Set the corresponding region interval to the “upper
   interval”
10: $succ(r_{cd}) \longleftarrow succ(r_i) \bigcup oneSuccessor$;
11: end
12: Return $succ(r_{cd})$;
13: END
```

present a *dynamic region encoding* scheme to capture this “context” information during the query processing time. In our solution, once a node receives its code from one of its predecessors, it obtains all the necessary information to behave correctly.

Definition 2. *(Dynamic Region Encoding) Given query region Q_{ab}, let x be the top-right point of master node $M(Q_{ab})$’s CAN zone. For each d-dimensional region $r \in RS(Q_{ab})$, we assign a d-digit code $ID(r)$ to region r. where $ID(r)[i]$ equals to ‘0’ if the interval of r on the i^{th} dimension = $[a_i, x_i]$; $ID(r)[i]$ = ‘1’ if the interval of r on the i^{th} dimension = $[x_i, b_i]$; $ID(r)[i]$ = ‘*’ if during the region partition the original interval on i^{th} dimension is not divided.*

Informally, the i^{th} digit of $ID(r)$ encodes whether r takes on the “lower half” (‘0’), the “upper half” (‘1’) or the “original interval” (‘*’) as the result of the corresponding region partitioning. Based on this region coding scheme, we define a “Skyline Precede” relation as follows:

Definition 3. *(Skyline Precede, $\prec$) Relation “Skyline Precede” ($\prec$) is a relation over region set $RS(Q_{ab})$: region r_i “Skyline Precede” r_j, or $r_i \prec r_j$, i.f.f. code $ID(r_i)$ differs from $ID(r_j)$* in only one *bit, say, the k^{th} bit, where $ID(r_i)[k]$ = ‘0′ and $ID(r_j)[k]$ = ‘1′. We denote $pred(r_i)$ as the set containing all the regions that “Skyline Precede” r_i, and $succ(r_i)$ as the set containing all the regions that r_i “Skyline Precede”.*

“Skyline Precede” precisely defines the order in which a distributed skyline query should be propagated and executed. In Algorithm 1 we describe how a specific region r_{cd} calculates its successor set $succ(r_{cd})$ in $RS(Q_{ab})$ based on its code $ID(r_{cd})$ ($pred(r_{cd})$ is computed analogously). Basically, each successor is generated by flipping one single ‘0’ bit to ‘1’ (`line 8`) and adjust the region interval on that dimension to the “upper interval” accordingly (`line 9`). Therefore, the query coordinates a, b of region Q_{ab}, and c, d (its own region r_{cd}) and code $ID(r_{cd})$ are all the information that needs to be sent for correct query propagation. Figure 3(a) illustrates the region codes and the “Skyline Precede” relationship on a 2-d CAN network given the initial query range ((0.3,0.3),(0.9,0.9)). For example, node 5 is given code ‘10’, its own query region ((0.5, 0.3),(0.9,0.5)), the whole query region ((0.3,0.3),(0.9,0.9)), it flips the ‘0’ bit to ‘1’ and adjust the y-interval from (0.3,0.5) to (0.5,0.9) and get its only successor region ((0.5,0.5), (0.9,0.9)) with code ‘11’.

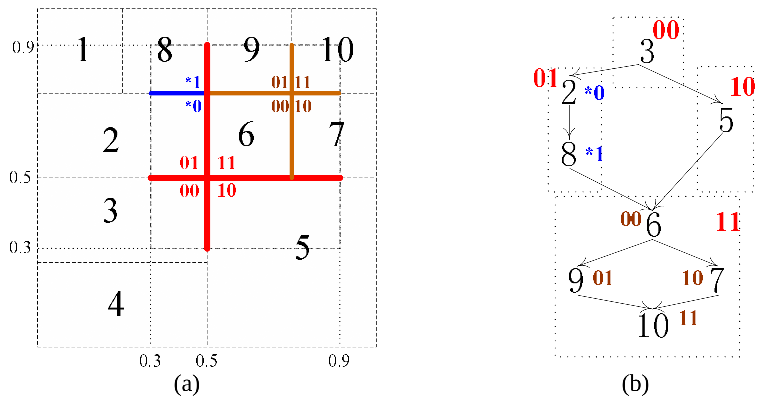


Fig. 3: (a) Finding the skyline points in range ((0.3,0.3),(0.9,0.9)). In total, region partitioning operation happens 3 times. (b) The query propagation order according to DSL.

The relationship between “Skyline Dependent, $\ll$” and “Skyline precede, $\prec$” is described by Lemma 1. Lemma 1 guarantees that, if we enforce that the query propagation follows the “Skyline Precede,$\prec$” relation, by the time a region starts, all and only its “Skyline Dependent, $\ll$” regions are completed.

Lemma 1. *For any two region r_i and r_j ($r_i, r_j \in RS(Q_{ab})$), $r_i \ll r_j$, i.f.f. there exists a sequence of regions, s.t.:$r_j \prec r_{j+1} \ldots \prec r_{i-1} \prec r_i$.*

Proof: According to Definition 2, in order for region r_i to be Skyline Dependent on region r_j, for those bits in which $ID(r_i)$ differs from code $ID(r_j)$, $ID(r_i)$ must be ‘1’ and $ID(r_j)$ must be ‘0’. This, together with Definition 3, ensures the correctness of Lemma 1. □

3.4 Algorithm Description

Now we present our system for **D**istributed **S**ky**L**ine query, or DSL. DSL uses the Content Addressable Network (CAN) as its underlying routing layer. We assume that the data is injected into the system either by feeds from merchant’s product databases [1] or from a Web database crawler that “pulls” structured data records from external Web sources [24]. The data space is normalized to $[0, 1]$ on each dimension and every data object is stored at the corresponding CAN node. Starting from the global query region, DSL recursively applies the region partitioning process to match the underlying CAN zones and the query propagation between the resulting subregions strictly complies with the “Skyline Precede” relationship which is enforced using dynamic region coding.

On each node involved, the DSL computation is composed of two asynchronous procedures: QUERY and COMPLETE. These two procedures are described in Algorithm 2. To activate a subregion Q_{cd} of Q_{ab}, a query message q is routed towards point c in the CAN virtual space using the CAN overlay routing mechanism. The node hosting c becomes the master node of the region, or $M(Q_{cd})$. Upon receiving q, the QUERY procedure on $M(Q_{cd})$ is invoked. DSL’s QUERY procedure on $M(Q_{cd})$ will be provided 4 parameters: 1) its region code $ID(Q_{cd})$; 2) its own query region Q_{cd}; 3) the skyline point set $skyline$ discovered from its “upstream” regions and 4) its “parent” query region Q_{ab}. Using this information, $M(Q_{cd})$ is able to calculate its position in the parent

Algorithm 2 Distributed Skyline(DSL) Computation

```
 1: $Q_{cd}$: current region to evaluate;        $Q_{ab}$: the "parent" region of $Q_{cd}$
 2: $ID(Q_{cd})$: region code for $Q_{cd}$;       $skyline$: skyline results from upstream regions;
 3: $M(Q_{cd})$: master node of $Q_{cd}$;
 4:
 5: QUERY($Q_{cd}$, $Q_{ab}$, $ID(Q_{cd})$, $skyline$)
 6: **Procedure**
 7: calculate predecessor set $pred(Q_{cd})$ and successor set $succ(Q_{cd})$;
 8: **if** all regions in $pred(Q_{cd})$ are completed **then**
 9:    **if** $skyline$ dominates $Q_{cd}$ **then**
10:       $M(Q_{cd})$.COMPLETE();
11:    **end if**
12:    $localresults \longleftarrow M(Q_{cd}).CalculateLocalSkyline(skyline, Q_{cd})$;
13:    $skyline \longleftarrow skyline \cup localresults$;
14:    **if** $M(Q_{cd}).zone$ covers $Q_{cd}$ **then**
15:       $M(Q_{cd})$.COMPLETE();
16:    **else**
17:       $M(Q_{cd})$ partitions $Q_{cd}$ into $RS(Q_{cd})$;
18:       **foreach** successor $Q_{gh}$ **in** $RS(Q_{cd})$
19:         $M(Q_{gh})$.QUERY($Q_{gh}$,$Q_{cd}$,$ID(Q_{gh})$,$skyline$);
20:    **end if**
21: **end if**
22: **End Procedure**
23:
24: COMPLETE()
25: **Procedure**
26: **if** $succ(Q_{cd})$ equals to NULL **then**
27:    $M(Q_{ab})$.COMPLETE();
28: **else**
29:    **foreach** successor $Q_{ef}$ **in** $succ(Q_{cd})$
30:      $M(Q_{ef})$.QUERY($Q_{ef}$, $Q_{ab}$, $ID(Q_{ef})$, $skyline$);
31: **end if**
32: **End Procedure**
```

query region Q_{ab}, i.e. its immediate predecessors $pred(Q_{cd})$ (`line 9`) and successors $succ(Q_{cd})$ (`line 10`). $M(Q_{cd})$ starts computation on its own query region Q_{cd} *only* after hearing from *all* its predecessors in $pred(Q_{cd})$ (`line 11`). $M(Q_{cd})$ first checks whether its own zone covers region Q_{cd} or whether Q_{cd} has already been dominated by "upstream" skyline points in $skyline$ (`line 14`). If either is positive, $M(Q_{cd})$ will not further partition its query region Q_{cd} and just directly call its local COMPLETE procedure meaning it finishes evaluating the region Q_{cd} (`line 15`). Otherwise it recursively partitions Q_{cd} into a new region set $RS(Q_{cd})$ (`line 17`), in which $M(Q_{cd})$ is responsible for the "first" subregion. For each successive region Q_{gh} in the new region set of $RS(Q_{cd})$, $M(Q_{cd})$ activates Q_{gh}'s QUERY procedure by routing a query message q' to the corresponding bottom-left virtual point g (`line 18-19`).

In COMPLETE procedure, $M(Q_{cd})$ proceeds with the computation by invoking the QUERY procedures on its successors in $succ(Q_{cd})$(`line 29-30`). If Q_{cd} happens to be the last subregion in region set $RS(Q_{ab})$, i.e. set $succ(Q_{cd})$ contains no successive regions. $M(Q_{cd})$ will pass the control back to the master node $M(Q_{ab})$ of its "parent" region Q_{ab} and invokes the COMPLETE procedure on $M(Q_{ab})$(`line 27`), i.e. the recursion “rebounds”. The entire computation terminates if the COMPLETE procedure on the master node of the global query region is invoked.

Figure 3(a) shows the recursive region partitioning process and its corresponding region codes of a constrained skyline query with initial query range $((0.3, 0.3), (0.9, 0.9))$. Figure 3(b) illustrates the actual query propagation order between machines according to DSL.

Theorem 2. *(Correctness and Progressiveness): For any constrained skyline query, DSL described above can progressively find all and only the correct skyline points in the system.*

3.5 Algorithm Analysis

In this subsection, we present two analytical results. First, in Theorem 3, we show DSL’s bandwidth behavior, which measures the inter-machine communication overhead and is critical for the system scalability. Then we show in Theorem 4 DSL’s optimality in terms of the total number of local skyline query invocation on each participating machine, which measures the I/O overhead and is important for its response time performance.

Theorem 3. *(Bandwidth Behavior): In DSL, only the data tuples in the final answer set may be transmitted across machines.*

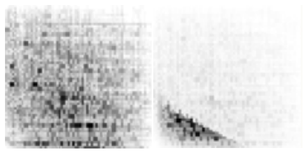
Proof: Let us consider a tuple d, s.t. d is not in the final skyline result set. Then there exists some skyline point d' that dominates d. Because DSL strictly follows the “Skyline Precede” order (`line 8`), according to Lemma 1, the region holding d' is evaluated before d’s region and there exists a “path” that connects these two. In other words, the machine that stores d must see d' in the $skyline$ set before calculate its local skyline, therefore d must be pruned and will not be transmitted. $\Box$

Theorem 4. *(Optimality): For a given data partitioning strategy, the total number of local skyline query invocations in DSL is minimized.*

Proof: First, it is trivial to show that for any query, DSL will only invoke the procedure $CalculateLocalSkyline()$ at most once on any participating node. Second, we show that if a node’s data region is dominated by some skyline points then DSL skip its local query procedure. Let us assume that the above theorem does not hold. Then there must exist a node n s.t. the data region (zone) of node n is dominated by some skyline point d (***i.e.*** n cannot have any points in final skyline result set) and the local procedure $CalculateLocalSkyline()$ has been invoked on n (`line 12`). Since DSL strictly follows the “Skyline Precede” order (`line 8`), node n must see d in $skyline$ set according to Lemma 1 and thus should have failed at `line 9`. In other words, `line 12` will not be executed on n. Contradiction. $\Box$

4 Load Balancing

Load balancing plays an important role in the performance of any practical distributed query system. Our system imposes two types of loads on each node: the data load and the query load. Some data load balancing techniques are described in [13], and specific load balancing work for CAN-based systems can be found in [15]. Here we focus only on addressing the query load imbalance issue inherent in DSL. We assume that compared to local query processing involving disk I/O, control messages consume negligible amounts of system resources. Therefore, our primary goal is to balance the number of local skyline queries processed on each node.


(a) original load distribution(independent and anticorrelated; random query).

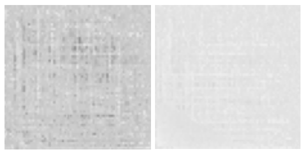

(b) after zone replication (independent and anti-correlated; random query).

Fig. 4: Query Load Visualization.

4.1 Query Load Imbalance in DSL

Our DSL solution leads to a natural query load imbalance. In DSL, query propagation always starts from the bottom-left part of the query box. An intermediate master node will not split its region if the region is dominated by “upstream” skyline points. When the region split does not take place, all nodes inside the region other than the master node will be left untouched which causes query load imbalance. Intuitively, for a given query range, nodes from the top-right regions are less likely to be queried than their “upstream” counterparts. In addition, real world query loads are more likely to be skewed, i.e. some query ranges are far more popular than others, which may further exacerbate this problem.

Figure 4(a) visualizes the original query load in a 2-d CAN space without load balancing. The darkness level of each zone represents the number of times a local skyline calculation is invoked on the corresponding node. The darker a zone appears, the heavier its load. We use independent as well as anti-correlated data sets, both with cardinality of 1 million on a 5000 node system. The workload consists of 1000 constrained skyline queries with randomly generated query ranges (For more about experiment setting, please see Section 5). We see in Figure 4(a) that the query load exhibits strong imbalance among nodes. In addition, zones at the bottom-left corner tend to be much more heavily loaded across both data sets.

4.2 Dynamic Zone Replication

To address the load imbalance problem, we propose a *dynamic zone replication* scheme. Our proposal is similar to the approach used in [26], but is tailored specifically to address the load imbalance in DSL.

Specifically, each node p_i in the system periodically generates m random points in the d dimensional CAN space. We set m equal to 10 by default. p_i routes probes to these points to ask for the query load at the local node. After obtaining all the replies, p_i compares its own load with the “random” probes. p_i will *only* initiate the zone replication process when its load is heavier than some threshold T of all samples. T is a system parameter set to 0.5 by default. In zone replication, p_i sends a copy of its zone contents to the least loaded node p_{min} in the sample set, and records p_{min}'s virtual coordinates in its $replicationlist$.

If a node has performed zone replication, local query processing adjusts to take advantage. When calculating a local skyline query, p_i picks a virtual coordinate v_j from its $replicationlist$ in a round-robin fashion. Then p_i forwards the query to the node p_j

Parameter	Domain	Default
Total nodes (Simulation)	[100,10000]	5000
Total nodes (Deployment)	80	80
Data cardinality	1M	1M
Dimensions (Simulation)	2, 3, 4, 5	2
Dimensions (Deployment)	2, 3, 4, 5	3
Query Range Pattern	random, biased	random

Table 1: Default setting

responsible for v_j for actual processing. To avoid unnecessary load probing messages, we set the probing interval proportional to the rank of node's query load in its latest samples. By doing so, lightly loaded machines probe less frequently while nodes in heavily loaded zones probe and distribute their load more aggressively. Figure 4(b) visualizes the system load distribution on both data sets after dynamic zone replication. On both data sets, the load distribution is much "smoother" than in Figure 4(a).

5 Performance Evaluation

5.1 Experimental Setup

We evaluate our DSL system through both simulation and measurements of a real deployed system. Our system is implemented on the Berkeley PIER query engine [16], and uses PIER's CAN implementation and runtime environment. We extended PIER's query capability by implementing the skyline operator as a plug-in operator. Because PIER uses identical interfaces for both discrete-event simulations and real deployment code, we used identical code in our simulations and cluster deployment. Our simulations ran on a Linux box with an Intel Pentium IV 2.4 GHz processor and 2 GB of RAM. The real measurement ran on a cluster composed of Dell PowerEdge 1750 Servers, each with Intel 2.6Ghz Xeon CPUs and 2GBs of memory.

We summarize default parameters in Table 1. Adopting the standard methodology in the literature, we use both independent (uniform) and anti-correlated data sets with cardinality of 1 million and dimensionality from 2–5 [19]. For those experiments where the results reflect the same trend on both data sets, we only show one of them to save space. The number of nodes in the simulation varies from 100 to 10000. The default query load in simulation consists of 1000 sequential constrained skyline queries with the query range randomly generated.

The real deployment runs on a cluster of 20 servers each running 4 node instances, for a total of 80 nodes. Since the cluster may be shared with other competing applications during these tests, we ran the same experiment 10 times and report the average response time below to provide a fair and reliable measurement. In addition, since the choice of local skyline processor is orthogonal to our comparison of different distribution methods, we use a simple Java implementation on all nodes. While this is sufficient to investigate DSL's query response behavior, the actual query delay reported below can be considerably improved using the "state of the art" centralized implementation [19] on each node.

We used three metrics in the experiment: percentage of nodes visited per query, number of data points transmitted per query and average query response time. All the

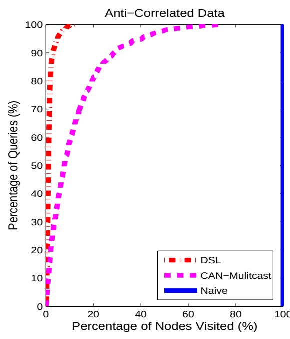


Fig. 5: Scalability comparison among all methods.

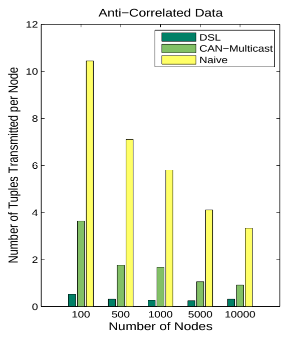


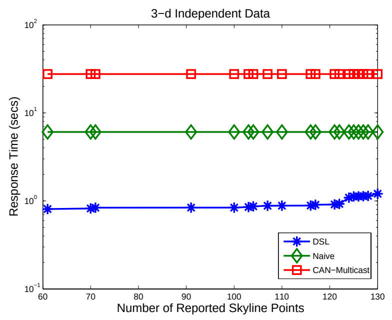
Fig. 6: Bandwidth comparison among all methods.

response time results are from real measurement and the results of the other two metrics are based on simulation. Our experiments are divided into two groups: First, we conduct comparative study on the three distributed skyline query methods described in this paper: the naive method, CAN-Multicast method, and DSL. All three algorithms use an identical simple local skyline query processor. Second, we study the effects of different system parameters on DSL's performance.

5.2 Comparative Studies

Scalability Comparison Figure 5 compares the three methods in terms of the number of node visited for each query on the anti-correlated data set. We show the cumulative density function (CDF) of percentage of queries (y-axis) against the percentage of nodes visited (x-axis). As expected, the naive method contacts all the nodes in the system for every incoming query, significantly limiting its scalability. The CAN-multicast method considerably improves upon the naive method: 90 percent of the queries will contact less than 40% of the nodes in the system. However, the remaining 10% of the queries still visit roughly 60% of all nodes. In a 5000 node system, this translates into visiting 3000 nodes for a single query! In contrast, DSL does a much better job of isolating the relevant data: no query involves more than 10% of all nodes, and roughly 90% of queries contact less than 1% of all nodes. This validates our claim that DSL prunes a large portion of of the data records during query propagation and only contacts the nodes and records that are necessary.

Bandwidth Comparison Figure 6 shows the bandwidth performance of all three methods on the anti-correlated data set when varying system size from 100 to 10000 nodes. We measure for each query the average number of data tuples transmitted per node. The x-axis plots the total number of nodes in the system and y-axis shows the average number of data points transmitted for a single query divided by the total number of nodes. Here we use the number of data points to characterize the bandwidth consumption, because data points are the dominant factor in the overall bandwidth usage when compared to control messages. For all system sizes, DSL outperforms the other two methods by one order of magnitude. This validates our claim in Theorem 3 that DSL saves bandwidth by transmitting only the data points inside the final result set.



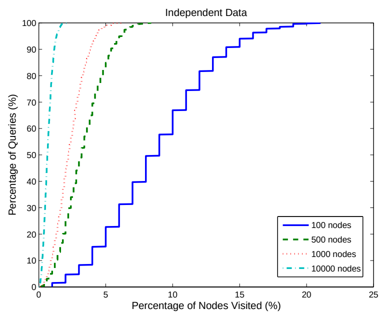


Fig. 7: Query response time comparison across three methods.

Fig. 8: Effects of system size on scalability in DSL.

Query Response Time Comparison We compare DSL's query response latency with the naive method and the CAN-multicast method. Due to the space constraints, we only show a workload containing one global skyline query, *i.e.* find all skyline points in the entire data set [3]. There are a total of 130 skyline points in the result set. For each skyline result reported, Figure 7 compares the response times among all three methods. Since the naive and CAN-multicast methods do not support progressiveness, their reported data points all have the same response delay (displayed as a flat line). The progressive behavior of DSL is not clearly reflected here due to the logarithmic scale of the y-axis. In fact, the initial results of DSL are reported within 0.8 seconds, while the last skyline point is reported in less than 1.2 seconds. As expected, DSL demonstrates orders of magnitude performance improvement over the other two methods.

A surprising result is that CAN multicast method performs much worse than the naive method. There are two reasons for this. First, our query is a global skyline query without constraints. This means the CAN-multicast method has no advantage over the naive method, because both methods need to contact all the nodes in the system. Second, the CAN-multicast method dynamically constructs a query propagation tree which is less efficient than the naive method where the entire tree is statically built at the beginning.

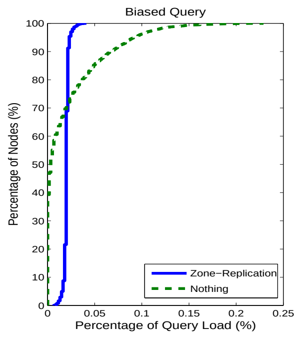
In summary, DSL is the clear winner over the other two alternative distribution methods in all the three metrics.

5.3 Performance Study of DSL

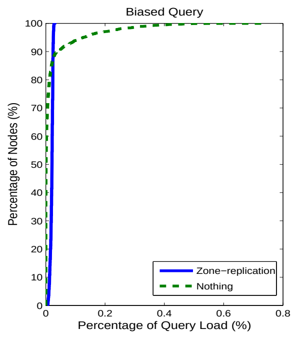
In this subsection, we focus on studying the effects of different system parameters on DSL's performance.

Effects of System Size Figure 8 uses a CDF to illustrate the effect of system size on the number of nodes visited per query. We ran the simulation under four different system settings consisting of 100, 500, 1000,10000 nodes respectively. Figure 8 shows a very clear trend: when the system size increases from 100 to 10000, the average number of participating nodes is quite stable. For example, when the system size is 100, most

[3] We have also tested several other query ranges and DSL is the consistent winner with the first several skyline points returned almost instantly



(a) independent data



(b) anti-correlated data

Fig. 9: Effectiveness of dynamic zone replication.

queries touches 15 machines. In contrast, when the system size grows to 10000, all queries involve less than 3% of the node population; and among them, a large portion (80%) of the queries only touch less than 0.5% (or 50 in a 10000 node system) of the nodes. The reason behind this is intuitive: with the virtual space partitioned by more nodes, the average zone size of a CAN node becomes smaller. The smaller granularity of zone partitioning allows a more accurate pruning of relevant nodes and data points. This simulation validates that DSL can scale up to a fairly large number of machines.

Effects of Dynamic Zone Replication on Load Balancing In this simulation, we study the effects of dynamic zone replication scheme on load balancing. We tested both the random query pattern as well as the biased query pattern. While a random query pattern generally yields more balanced query load, we only show the biased query load results because random query results were already visualized in Figure 4.

Figure 9 compares query load distributions before and after dynamic zone replication. The workload consists of 1000 constrained skyline queries evaluated on both anti-correlated and independent data sets. Each node reports its local query load during the whole process in terms of the number of times its local skyline procedure is invoked. The x-axis represents the query load percentage and the y-axis plots the percentage of nodes with that load. With 5000 nodes in the system, a node in a perfectly balanced system would perform 0.02% of the total number of local query operations in the system.

The original load distribution is clearly imbalanced. In the anti-correlated data set, 90% of the nodes in the system have negligible query load while the 10% are heavily loaded. 2% of the nodes are each responsible for more than 0.2% of the total query load, or overloaded by a factor of 10! These are the “upper stream” nodes residing in the bottom-left part of the virtual space. After dynamic zone replication is used on both data sets, the query load is much more evenly distributed and closer to the ideal. Together with the previous visualization results, these results clearly show that dynamic zone replication is effective for balancing the query load in the system.

Effects of Dimensionality on Bandwidth Now we study the effects of dimensionality on DSL’s bandwidth overhead. We vary the dimensionality of queries from 2 to 5,

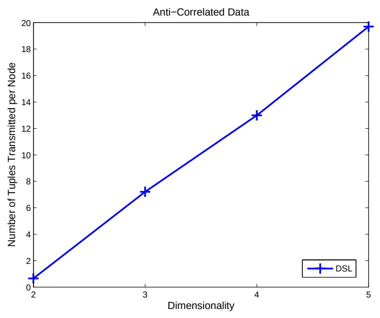


Fig. 10: Effects of dimensionality on bandwidth in DSL.

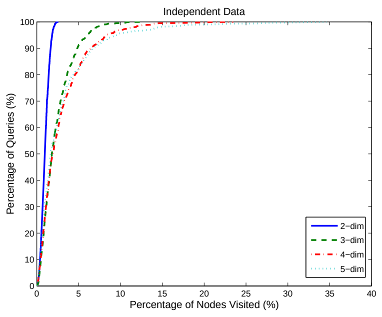


Fig. 11: Effects of dimensionality on scalability in DSL.

which according to [8] satisfies most real world applications involving multi-objective decisions.

Figure 10 shows the effect of dimensionality on the average bandwidth usage on the anti-correlated data set. The y-axis represents the average number of data points transmitted by every node for each query and the x-axis plots the dimensionality. Overall, the bandwidth usage steadily increases with the dimensionality. Specifically, on a 2-d data set, an average node only injects 1 data point into the network and this number grows to 20 on the 5-d data set. The main reason for this increase is that the original size of the skyline result set increases rapidly with dimensionality and thus more result points need to be transmitted with the query message from the “upstream” machines to the “downstream” nodes, leading to greater bandwidth consumption.

Effects of Dimensionality on Scalability Figure 11 shows the effects of dimensionality on the percentage of nodes visited per query on the independent data set. We vary the dimensionality from 2 to 5 and show the relationship between the query load percentage and node percentage involved. With the increase of the dimensionality, more nodes are involved in query processing. This is due to two reasons. First, as described above, with the increase in dimensionality, the skyline result size increases dramatically and thus more nodes are likely to store data points in the final result set. Second, with higher dimensionality, more virtual space needs to be visited while the number of machines used to partition the virtual spaces remains the same. Note that even when the dimension number grows to as large as 5, most queries are evaluated across a small portion of the nodes. Specifically, more than 90% of queries each require less than 10% of all nodes. This demonstrates that DSL is scalable with the increase of dimensionality.

Effects of Dimensionality on Response Time In Figure 12, we study the effects of dimensionality on the query response time. We use the independent data set while varying the dimensionality from 2 to 5. In this experiment, we still use one global skyline query as our query load. Under each dimensionality setting, Figure 12 shows the average response delay for all skyline results reported by DSL. Due to the progressiveness of DSL, initial result points are typically received much faster than this average number.

As the number of dimensions grows, the average delay increases steadily. On the 2-d data set, the average response delay is 0.6 seconds. As the number of dimensions

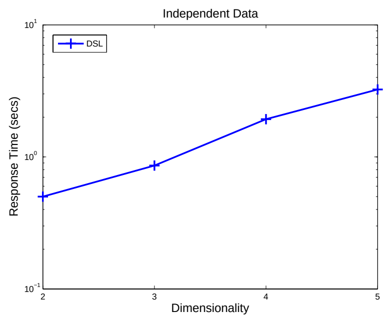


Fig. 12: Effects of dimensionality on DSL response time for a 2-d data set.

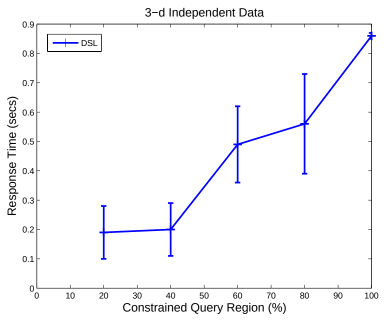


Fig. 13: Effects of query box size on response time.

grows to 5, the average response time grows to roughly 2 seconds. This is explained by three factors. First, as was shown in Figure 10, DSL's bandwidth consumption increases with dimensionality, and therefore more time is spent on data transmission between nodes. Second, as was shown in simulations (Figure 11), the percentage of nodes visited per query also increases in the higher dimensional data sets. Since more machines are involved, it takes more time for the query to propagate to the "downstream" nodes, further delaying the reporting of results from these machines. Finally, local skyline calculations at each node also becomes more expensive in higher dimension datasets.

Effects of Query Range on Response Time In Figure 13, we investigate the effects of query box size on the response time. For each query box size, we generate 10 constrained skyline queries, each of which has a query range covering a certain percentage of the entire data space. We show the average response delay of 10 queries. For each point along the line, we also draw a bounding box that depicts the standard deviation of the response delay. In the case of 100% query range, the standard deviation is very small. Clearly, the average response delay increases with the growth of the query box size. In particular, when the query range equals 20%, the average delay is less than 0.2 seconds. The delay increases to 0.85 seconds when the query range grows to 100%.

6 Related Work

Skyline query processing algorithms have received considerable attention in recent database research. Early work [8] proposed the Block-nested loops, Divide and conquer, and B-tree solutions. Later work proposed the Indexing and Bitmaps solutions [23]. This was further improved in [17], where Nearest neighbor search (*NN*) was used on a R-tree indexed data set to progressively discover skyline points. The best centralized method, *BBS* [19], has been shown to be I/O optimal, and outperforms *NN*. Other work addresses continuous skyline queries over data streams [18], extends skyline query to categorical attribute domains where total order may not exist [9]. One latest work [14] introduces a new generic algorithm `LESS` with $O(n)$ average case running time. Similar results are presented in [25] and [20] on efficient computation methods of finding skyline points in subspaces through exploiting various sharing strategies.

The notion of recursive partitioning of the data space in DSL is similar to *NN* and *BBS*. However, fundamental differences distinguish our effort from these two works. In *NN*, the order of the intermediate partitions will not influence its correctness since the nearest neighbor query guarantees to find the lower left skyline point each time. In DSL, the lower left partition may not contain any data points at all. On the other hand, unlike *BBS*, there does not exist an “oracle” in the distributed environment to order the candidate partitions in a centralized priority queue. Moreover, DSL partitions the query region during run-time to match the underlying node zones, since, unlike *BBS*, there does not exist certain *a-priori* “recursive” index structures like R-Tree. Therefore, the novelty of DSL mainly lies in the exploitation of the partial order over data regions and the dynamic region partitioning and encoding schemes to enforce this order at runtime in a share-nothing architecture.

The only previous work that calculates skyline query over distributed sources was presented in [6]. In this work, data is vertically distributed across different web information services with each site providing one attribute of the data object. Skyline points are calculated per site and reported to users at a central point. This can limit the scale of distribution. In contrast, we horizontally partition data across different machines, *i.e.* each machine stores a subset of the entire data record set. Unlike the previous approach, our solution provides incremental scalability, where performance is improved by adding additional machines to the cluster. Our system automatically balances load by distributing objects to the new node.

Parallel databases [10] and distributed database systems such as Mariposa [22] used multiple machines to efficiently process queries on partitioned data relations. In particular, previous research on parallel database systems have shown that “range partitioning” can successfully help query processing in share-nothing architectures(e.g. parallel sorting [11] and parallel join [12]). Skyline processing in these settings has not been studied, and is the problem addressed in this paper.

7 Conclusion and Future Work

In this paper, we address an important problem of parallelizing the progressive skyline queries on share nothing architectures. Central to our DSL algorithm is the use of partial orders over data partitions. We propose two methods, recursive region partitioning and dynamic region encoding, to implement this partial order for pipelining machines in query execution. We provide an analytical result that shows our algorithm to be optimal in minimizing local queries. Finally, we introduce the use of dynamic zone replication to distribute computation evenly across nodes.

We implemented the DSL system on top of the PIER code base, and used the result to perform extensive experiments on a simulation platform and a real cluster deployment. Our evaluation shows DSL to significantly outperform other distribution approaches, and that dynamic zone replication is extremely effective in distribution query load. As future work, we will further explore the resilience of query processing to node failures and replications, and DSL’s bandwidth consumption in higher dimension data sets.

References

1. Froogle data feeds. https://www.google.com/froogle/merchants/feed_instructions.html.
2. Yahoo! automobile. http://autos.yahoo.com/.
3. Yahoo! real estate. http://realestate.yahoo.com/.
4. T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW (network of workstations). *IEEE Micro*, 15(1):54–64, Feb 1995.
5. A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. In *Proc. of SIGMOD*, Tucson, AZ, May 1997.
6. W. Balke, U. Guntzer, and X.Zheng. Efficient distributed skylining for web information systems. In *Proc. of EDBT*, 2004.
7. L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, March/April 2003.
8. S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. of ICDE*, 2001.
9. C.-Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *Proc. of SIGMOD*, 2005.
10. D. Dewitt and J. Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6), 1992.
11. D. Dewitt, J. Naughton, D. Scheneider, and S. Seshadri. Parallel sorting on a shared-nothing architecture. 1991.
12. D. Dewitt, J. Naughton, D. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. of VLDB*, 1992.
13. P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partioned data with applications to peer-to-peer systems. In *Proc. of VLDB*, 2004.
14. P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *Proc. of VLDB*, 2005.
15. A. Gupta, O. D. Sabin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish/subsribe over P2P networks. In *Proc. of Middleware*, 2004.
16. R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *Proc. of VLDB*, 2003.
17. D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: an online algorithm for skyline queries. In *Proc. of VLDB*, 2002.
18. X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *Proc. of ICDE*, 2005.
19. D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proc. of SIGMOD*, 2003.
20. J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *Proc. of VLDB*, 2005.
21. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. of SIGCOMM*, Aug 2001.
22. M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1), 1996.
23. K. L. Tan, P. K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proc. of VLDB*, 2001.
24. P. Wu, J.-R. Wen, H. Liu, and W.-Y. Ma. Query selection techniques for efficient crawling of structured web sources. In *Proc. of ICDE*, 2006.
25. Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proc. of VLDB*, 2005.
26. Y. Zhou, B. C. Ooi, and K.-L. Tan. Dynamic load management for distributed continous query systems. In *Proc. of ICDE*, 2005.