

# Impression Store: Compressive Sensing-based Storage for Big Data Analytics

Jiaxing Zhang, Ying Yan, Liang Jeff Chen, Minjie Wang, Thomas Moscibroda, Zheng Zhang  
*Microsoft Research*

## Abstract

For many big data analytics workloads, approximate results suffice. This begs the question, whether and how the underlying system architecture can take advantage of such relaxations, thereby lifting constraints inherent in today’s architectures. This position paper explores one of the possible directions. *Impression Store* is a distributed storage system with the abstraction of big data vectors. It aggregates updates internally and responds to the retrieval of top- $K$  high-value entries. With proper extension, *Impression Store* supports various aggregations, top- $K$  queries, outlier and major mode detection. While restricted in scope, such queries represent a substantial and important portion of many production workloads. In return, the system has unparalleled scalability; any node in the system can process any query, both reads and updates. The key technique we leverage is compressive sensing, a technique that substantially reduces the amount of active memory state, IO, and traffic volume needed to achieve such scalability.

## 1 Introduction

Data volume has been increasing at an unprecedented pace. Systems for storing and querying big data have developed from a single server to thousands of commodity machines, with no end in sight to this growth. Such drastic changes bring many challenges to system design; deep complexities are baked into the ever-increasing cloud of machines. Yet, performance is often lame, limiting both the system’s and the data’s usefulness.

For example, an analysis of hundreds of daily in-production data mining jobs on Microsoft’s SCOPE [7, 22] platform—a large-scale data processing engine that combines MapReduce’s explicit parallel executions and SQL’s declarative query constructs—reveals that they consume two thousand machine-hours on average, and some jobs take up to 10 hours. Such performance numbers highlight the limits of today’s brute-force, heavy

provisioning approach; despite consisting of thousands of machines, applications can issue only a low volume of queries toward these systems.

In this paper, we investigate the benefits of a radically different approach. We postulate that, by and large, the complexity of today’s systems is a result of an architecture that neither anticipates nor accommodates the particular needs of big data analytics. There is a clear division of labor between storage and processing. The responsibility of storage is to store data reliably and to provide as much IO-bandwidth to the data as possible, while the one of processing is to handle query logics. However, applications are rarely interested in raw data records. Instead, they typically issue queries against aggregated values (e.g. `sum...group by...`), which can be understood by humans. Of most interest to users are analytics that “query impressions”, i.e., high-value statistics such as majority, mode, top- $K$  and outliers. For such queries, there is even more potential for reducing storage, IO-bandwidth requirements and communication. This is especially the case for high-dimensional data, a combination of numerous data attributes that can typically be represented in a sparse form at little loss of fidelity. Sparsity implies that when “querying impressions”, the majority of mass concentrates on the heads of the distribution, a characteristic that we can exploit for efficient processing.

In this paper, we show that for this category of big data analytics—cases such as top- $K$  analysis and outlier detection that constitutes a large portion of production workloads as we observed—new system-level abstractions can yield fundamental improvements in performance and scalability. Specifically, we adopt an application-driven approach to re-design an architecture, in which we intentionally blur the boundary of storage and processing. The insight is that we do not need to store all data accurately, as long as the system can return application-defined elite components successfully. This saves storage capacity and IO bandwidth, and allows for efficient and scalable parallel updates and queries.

Different from a traditional key-value store [8], *Impression Store* provides an abstraction of a big data vector that aggregates value updates to each element and only supports the retrieval of top- $K$  components. This abstraction may seem constrained, but supports a wide array of applications we have in mind. In return, we obtain a system that is simpler, more efficient and fundamentally more scalable. In particular, in *Impression Store* any node can process any query (including updates) concurrently within bounded response time. Since the delay between updates and queries is shrunk dramatically, the system can provide continuous snapshots against a standing query. As a result, our design can also be an alternative to scalable streaming processing, again with a radically different architecture.

One key enabling technology in our design is *Compressive Sensing* (CS). CS is an efficient technique of sampling high-dimensional data that has an underlying sparse structure. In the context of our work, we can regard CS as a technique that provides a compressed snapshot (or sketch) of data. CS is essentially a lossy compression, but the corresponding decompression algorithm discriminates components in data vectors by their values. Consequently, it is possible to prioritize the accuracy of *principal* components with high values. This property fits our target applications well, because elite components, such as top- $K$  components and outliers, can be represented as high values by appropriate modelling and/or transformations. Furthermore, we show that CS naturally supports increments on compressed data. Increments are commutative and associative, allowing for parallel processing at a massive scale. This property is important because applications’ target data, such as query or click logs, is continuously changing.

The system architecture of *Impression Store* is shown in Figure 1. Instead of storing the entirety or portions of the original data vector, each node in the store maintains a compressed snapshot of the complete data vector. These snapshots are obtained by CS-techniques with a compression ratio that is necessary to recover only the application-defined principal components. Queries are issued through the *Impression Store*’s API: A read-only query can be processed by any node over its local snapshot; an update query can be issued against the compressed snapshot in a random selected node. Updates are further aggregated, compressed, and propagated to other nodes following an anti-entropy protocol. These attributes ensure that the system’s throughput is essentially proportional to its scale. Since both the state stored on each node, as well as that of the anti-entropy protocol are compressed, storage, IO bandwidth requirements and communication overhead is drastically reduced. This reduction relieves the pressure on the underlying hardware infrastructure, making the system easier to scale.

## 1.1 Related Work

Many distributed stores are off the shelf [1, 8, 15] for general-purpose applications by keeping all the data precisely. In contrast, *Impression Store* is designed to serve statistical queries, which leads to more economic storage and communication.

Compressive Sensing (CS) is a breakthrough in the signal processing community. Its basic ideas and mathematical foundations have been established in [5, 6, 12, 18]. CS is an efficient approach to sample and reconstruct *sparse* data, with many applications (e.g., photography [13], image reconstruction and recognition [19], and network traffic monitoring [21]). In our work, we use CS to compress and recover sparse data vectors.

Traditional lossless and lossy compression techniques [2, 3, 10, 11, 23] have been developed for many decades. However, they are either inefficient for continuously-changing big data or only fit for some specific domains such as images and videos. Data sketches [9, 16] in databases provide compressed snapshots of big data. CS compressed data can be viewed as a special form of data sketches, with high accuracy on principal components.

## 2 Compressive Sensing for Big Data

We introduce Compressive Sensing (CS) and show why it is a good fit for “querying impressions”. A data vector  $\mathbf{x} = [x_1, x_2, \dots, x_N]^T \in \mathbb{R}^N$  is sparse if there are only a few high-value *principal* components over other minor ones. In *compression*, given an  $M \times N$  ( $M < N$ ) measurement matrix  $\Phi = [\varphi_1, \varphi_2, \dots, \varphi_N]$ , each of whose column vectors  $\varphi_i$  is randomly generated, the data vector  $\mathbf{x}$  is compressed to a measurement vector  $\mathbf{y}$  by  $\mathbf{y} = \Phi\mathbf{x}$ .

Due to the linearity in the compression, CS is uniquely suited for data that is (i) distributed in different locations, and (ii) updated frequently. When the global data vector  $\mathbf{x}$  is an aggregation of multiple local data vectors  $\mathbf{x}_l$  distributed on different nodes by  $\mathbf{x} = \sum_l \mathbf{x}_l$ , we first do the compression on each individual node and then aggregate local measurements together for the global measurement on  $\mathbf{x}$ . That is,  $\mathbf{y} = \Phi\mathbf{x} = \sum_l \Phi\mathbf{x}_l = \sum_l \mathbf{y}_l$ . Similarly, when there is an update vector  $\Delta\mathbf{x}$  updating  $\mathbf{x}$  to  $\mathbf{x} + \Delta\mathbf{x}$ , we can first compress the update and add that to the original measurement incrementally:  $\Phi(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{y} + \Phi\Delta\mathbf{x} = \mathbf{y} + \Delta\mathbf{y}$ . Overall, the linearity in CS allows us to avoid decompressing compressed data for aggregation or updates, scenarios in which traditional compression techniques often face challenges and may not be applicable.

To reconstruct the sparse data vector from the measurements, one of the most widely used recovery algorithms is Orthogonal Matching Pursuit (OMP) [17, 18], which is a greedy approach. Starting from a residual  $\mathbf{r} = \mathbf{y}$  and an empty selection set  $\Psi$ , OMP in each iteration selects the column vector  $\phi \in \Phi$  that has the largest

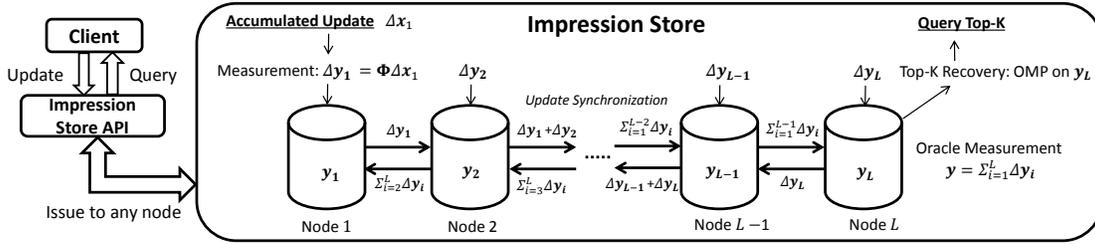


Figure 1: Architecture of *Impression Store*

inner product with the current residual  $\mathbf{r}$ , and inserts it into  $\Psi$ .  $\mathbf{y}$  is projected to the subspace spanned by  $\Psi$  to get its projection  $\mathbf{y}'$ , and the residual is updated by  $\mathbf{r} = \mathbf{y} - \mathbf{y}'$ . When OMP is finished, the projected coefficients are the estimation of the principal components in  $\mathbf{x}$ .

Due to its greedy nature, OMP has unique advantages when used on distributed big data, where measurement and recovery can pose significant IO-bandwidth, network traffic and computational overhead. OMP naturally recovers the top- $K$  components in the first few iterations. Thus, for many impression queries, OMP can be stopped early. Also, OMP requires much fewer measurements to recover the top- $K$  components, as compared to when recovering the entire data. Also, OMP can be extended to *Bias OMP* (BOMP) [20] to retrieve the majority (mode) and outliers from non-sparse data.

Figure 2 shows an experiment on the recovery of a large-scale production revenue data set with 12891 advertisement groups (see Section 4 for details). The measurement size  $M$  must be bigger than 800 to recover the top-50 components. However, if we are only interested in the top-20 components,  $M = 300$  is enough for high accuracy reconstruction using OMP. We observe that  $M \ll N$  (in our case  $M = 2.3\%N$ ) and  $K \ll M$  ( $K = 6.3\%M$ ) holds qualitatively for sparse data. This is the underlying reason for the performance savings of *Impression Store*.

### 3 Design

#### 3.1 Data Model and Architecture

The overall architecture of *Impression Store* is illustrated in Figure 1. It is the responsibility of the client of *Impression Store* to map a table into data vectors by flattening the interesting values. The keys for these values are organized into a dictionary that is persistently stored and cached on the client. A translation module in the client is responsible for translating the SQL-like queries to operations on these data vectors. For ease of exposition, we describe the main *Impression Store* APIs of one data vector  $\mathbf{x}$ . Other than miscellaneous operations such as setting up and destroying the vectors, there are two main operators: `Update(i, Δx)` adds or subtracts the value  $\Delta x$  at index  $i$  of  $\mathbf{x}$ . For retrieval, `Query(K)` returns the top- $K$  values in  $\mathbf{x}$ . The first operator mutates the vector entries, whereas the second one is a read-only

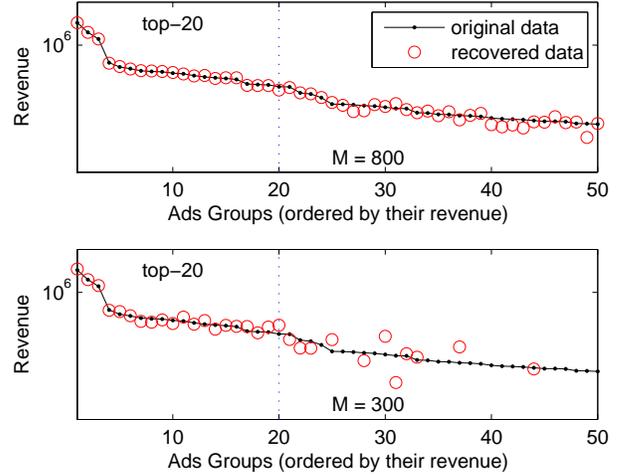


Figure 2: An example of OMP recovery for top- $K$  components. Only the top-50 components are shown. The two figures use different measurement size  $M$ .

query. Any of the queries can be issued to any node in the system. Using BOMP, *Impression Store* also provides `OutlierAnalysis(K)` to return the majority and top- $K$  outliers in  $\mathbf{x}$ .

Each node works on three tasks continuously: (1) **Aggregate and Compress Data Updates**. The accumulated client-side update  $\Delta \mathbf{x}_l$  results in the corresponding measurement,  $\Delta \mathbf{y}_l = \Phi \Delta \mathbf{x}_l$ .  $\Phi$  is an  $M \times N$  measurement matrix that all nodes agree on and can be generated on the fly. (2) **Update Synchronization**, to be described in more details shortly, propagates the changes throughout the system so that  $\mathbf{y}_l$  (the snapshot at  $l$ ) is as close to the global oracle measurement  $\mathbf{y} = \sum_l \Delta \mathbf{y}_l$  as possible. (3) **Top- $K$  Recovery**. When any node  $l$  receives `Query(K)`, it reconstructs the top- $K$  values from  $\mathbf{y}_l$  using OMP, and returns the index-value pairs.

Throughout the system, both the amount of state and traffic volume are bounded by  $O(M)$ , a small fraction of the original data size  $N$ . Compression and recovery takes computation power, but they involve only standard matrix computations that can be greatly accelerated with GPUs, which are themselves commodities. It is insightful to consider an alternative design that achieves the any-query-any-node scaling property by spreading updates across all nodes and then aggregating them. Such an alternative would require an  $N/M$ -fold increase of

state space and network bandwidth, but it could return the approximate value at any position (which our applications do not require), and perform top- $K$  queries with  $O(N \log K)$  computation complexity (non-trivial for large  $N$ ). Therefore, *Impression Store* can be viewed as a design that trades off computation for scalability.

One may argue that the above comparison is unfair, since *Impression Store* performs lossy compression, whereas the other alternative is more expensive simply because it pays the cost of being lossless. Such baseline can indeed be relaxed, by keeping  $M$  higher ranked entries and returning the top- $K$  among them. One problem is that it is unlikely that each node agrees on the position of the  $M$  entries to keep. Even if the distribution is uniform enough, not recording any information other than the  $M$  top entries runs the risk that some entries outside them may eventually catch up and enter the competition, a scenario that is not at all unlikely. The root of the problem is that in this case the baseline is losing too much. As we will show in Section 4, *Impression Store* can cope with such dynamics much more effectively.

### 3.2 Update Synchronization

In *Impression Store*, updates are randomly issued at any node. The compressed measurement of the client-side updates received at node  $l$ ,  $\Delta \mathbf{y}_l$ , needs to be propagated and aggregated. At any given point of time, there is a global oracle measurement  $\mathbf{y} = \sum_l \Delta \mathbf{y}_l$ , and the goal of the synchronization protocol is to let every node’s local version,  $\mathbf{y}_l$ , converge to  $\mathbf{y}$  quickly and eventually. A conventional approach is to perform in-network aggregation, deriving  $\mathbf{y}$  at the root of the tree, which is then propagated downwards. Generalizing that idea and eliminating the single root bottleneck, we describe an alternative design that works for any loop-free topology.

We use  $\mathcal{N}(l)$  to denote the neighbors of node  $l$ .  $\mathbf{y}_l$ , the local version of the global measurement at node  $l$ , is  $\mathbf{y}_l = \Delta \mathbf{y}_l + \sum_{q \in \mathcal{N}(l)} \Psi_{q \rightarrow l}$ . Intuitively,  $\Psi_{q \rightarrow l}$  represents the contribution of the neighbour  $q$  and its sub-tree from  $l$ ’s point of view. This state has the master copy at  $q$  and a mirror copy at  $l$ , denoted by  $\Psi_{q \rightarrow l}^q$  and  $\Psi_{q \rightarrow l}^l$  respectively. At quiescent time they are equal, and if the master copy  $\Psi_{q \rightarrow l}^q$  advances, anti-entropy between the two will update the mirror copy  $\Psi_{q \rightarrow l}^l$ . If  $\mathbf{y}_l$  is changed due to an update of  $\Delta \mathbf{y}_l$  or from its neighbours, then it re-computes  $\Psi_{l \rightarrow p}^l = \mathbf{y}_l - \Psi_{p \rightarrow l}^l$ , and triggers anti-entropy.

The protocol can be proved to achieve *eventual consistency*. Therefore all nodes in *Impression Store* will converge to the same measurement vector eventually. Since the only requirement of the protocol is loop-free topology, it is more flexible. A single-level tree with one node fans out to all others implements a master-slave architecture and converges in two hops, whereas a chain topol-

ogy (Figure 1) converges with  $O(L)$  speed but has even loads. Anything in between works equally well, except with different convergence and load-balance trade-offs.

### 3.3 Fault Tolerance

It is very expensive to prevent a distributed system from losing any updates when some nodes suffer permanent failure. It is *not* our design goal that *Impression Store* does not lose any updates. When a node  $l$  fails and is replaced by a new one, the major loss is the updates that are applied to  $l$  but not yet propagated to its neighbors. For the scenario we target, we believe such loss is tolerable. As long as the switch-over is rapid and update frequency high enough, the chance of messing up the rank is low.

Our failover protocol tries to recover various states on a single failed node  $l$ . We first restore the invariance  $\Psi_{p \rightarrow l}^l \equiv \Psi_{p \rightarrow l}^p$  and  $\Psi_{l \rightarrow p}^l \equiv \Psi_{l \rightarrow p}^p$  by copying the relevant states from every neighbor  $p$ . Then we recover  $\Delta \mathbf{y}_l$  by computing  $\Psi_{l \rightarrow p}^l - \sum_{\mathcal{N}(l)/p} \Psi_{q \rightarrow l}^l$  for each neighbor  $p$  and pick the one with largest version number as the restored  $\Delta \mathbf{y}_l$ . Finally, we re-compute all  $\Psi_{l \rightarrow p}^l$  and trigger the anti-entropy protocol to synchronize with their mirror copies.

### 3.4 Optimization

The computation complexity of the standard OMP algorithm is  $O(M^2 * N)$ , i.e., it is costly to recover the top  $k$  on a data vector with big size  $N$ . However, there has been work to accelerate the recovery algorithms by GPU with 30X  $\sim$  40X speed-up [4, 14]. Using an off-the-shelf device NVIDIA GTX480, OMP on a data vector with tens of thousands of components takes only about 1 second [14].

The OMP algorithm could be further optimized for continuously updated data. In our optimized OMP, it keeps remembering the positions of the selected column vectors. These position correspond to the principal components in current data vector  $\mathbf{x}$ , and do not change a lot over a short period. Then, our OMP prioritizes those positions for the next recovery with only  $O(M^3)$  complexity. Due to limitation of space, we skip more details.

## 4 Preliminary Results

In this section, we focus on evaluating recovery quality in *Impression Store* using simulation, driven by real product data of the revenue on advertisement (Ads) entries in Bing search engine. Updates of these entries are spread across different machines, and are grouped by the type of the hosting-page that is described by a tuple (Market, Vertical, QueryClass, . . .). After removing the trivial zeros, the aggregated revenues on all the Ads groups construct a data vector with size 12891. Taking a snapshot  $\mathbf{x}$  of the data vector, we test the top- $K$

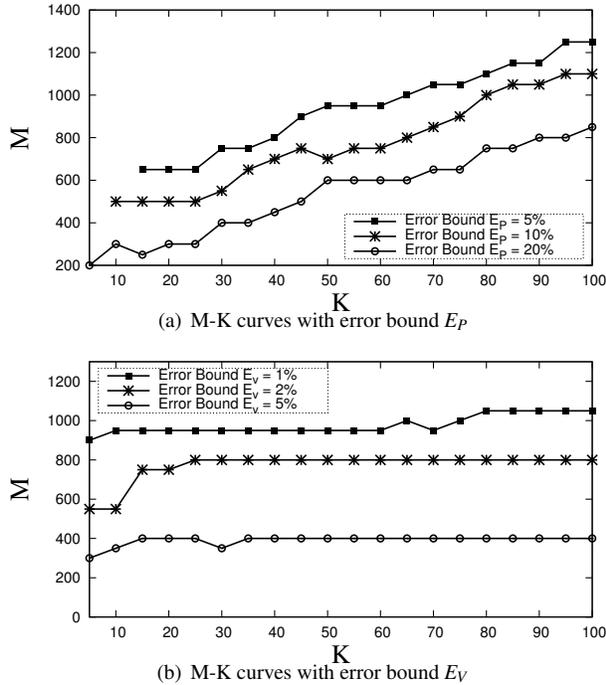


Figure 3:  $M$  v.s.  $K$  over different error bounds

recovery accuracy by comparing the ground-truth top- $K$  list  $I$  versus the recovered one  $\tilde{I}$ .

There are two errors that are of interests, captured by the following metrics:

**Error on Position** :  $E_P = 1 - |I \cap \tilde{I}|/K$ .

**Error on Value** :  $E_V = \|\tilde{\mathbf{x}}_I - \mathbf{x}_I\|_2 / \|\mathbf{x}_I\|_2$ .

Here,  $\mathbf{x}_I$  and  $\tilde{\mathbf{x}}_I$  are vectors of the entries specified by  $I$  and  $\tilde{I}$ , respectively.

We first evaluate the effect of measurement size  $M$  on the top- $K$  result quality. For each  $(M, K)$  pair, we run our compression and recovery algorithm 100 times, each with random measurement matrices; the maximum  $E_P$  and  $E_V$  of these runs are the error bounds. The effect on  $E_P$  is illustrated in Figure 3(a). We can see that,  $M$  increases roughly linearly with  $K$  to keep the same error bound. With  $E_P = 20\%$ , to get top-10 values, there needs only 300 measurements,  $2.3\% \times N$ . The effect on  $E_V$  are demonstrated in Figure 3(b). The accuracy of the recovered top- $K$  values is stable with the increase of  $K$  for a given  $M$ . When  $M = 3\% \times N$ , *Impression Store* can return top-100 values with  $E_V \leq 5\%$ .

Next, we evaluate the robustness of *Impression Store* with a fixed measurement size  $M$ . The setting is such that there is one entry below rank 300, which continuous to receive positive updates, whereas all other entries gets random updates. The correct behavior of the system should be that this entry is gradually captured, and gets increasingly accurate. We compare against a traditional store which keeps the top-300 list only, and

therefore an entry can enter the list if only if a one-time update bumps it into the top-300 list. As shown in Figure 4, after several updates, this entry gets captured by *Impression Store* and is recovered more and more accurately after  $t = 20$ . In contrast, in the traditional store, this entry makes it into top-300 only after one lucky update, and its prior history was completely forgotten, resulting in a constant error.

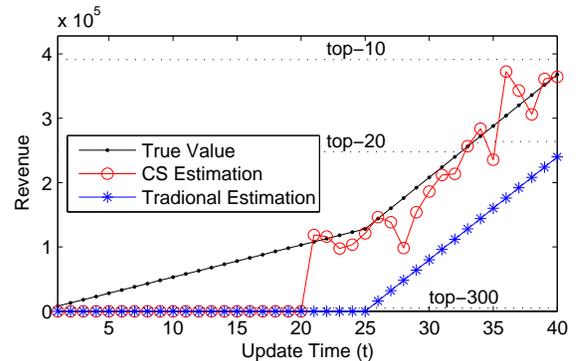


Figure 4: Estimation of increasing revenue on a Ads group

## 5 Conclusion and Future Work

Designing an efficient and scalable system infrastructure for massive-scale big data analytics is a tremendous challenge. While analytics algorithms are increasingly relying on sophisticated mathematical techniques to exploit the sparsity of the data, the underlying system infrastructure has largely remained unchanged. In this paper, we argue that there are substantial benefits in not only adapting data processing, but the entire system architecture. If data is sparse and queries are mostly focusing on dominant components, a distributed system architecture that tightly integrates with Compressive Sensing can fundamentally increase performance and scalability.

There are numerous avenues for future work. Our current prototype builds on a narrow subset of CS theory. By exploiting CS techniques more thoroughly, *Impression Store* can support more sophisticated queries. For example, it is possible to directly recover a linear transformation from the data vector, without changing the measurement. In this way, *Impression Store* could return aggregations on a key  $A$  from a table keyed on  $(A, B)$ .

Another extension is sampling. In the current design, the original data is not kept. However, the raw data on each node is effectively a sample of the entire data. These samples can be used to answer queries other than those covered by CS. Multiple parallel queries to different nodes can help to improve confidence.

## References

- [1] *Hadoop*. <http://hadoop.apache.org/>.
- [2] *JPEG*. <http://www.jpeg.org/>.
- [3] *MPEG*. <http://mpeg.chiariglione.org/>.
- [4] ANDREUT, M. Fast gpu implementation of sparse signal recovery from random projections. *Engineering Letters* 17, 3 (2009), 151–158.
- [5] CANDÈS, E. J., ROMBERG, J., AND TAO, T. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *Information Theory, IEEE Transactions on* 52, 2 (2006), 489–509.
- [6] CANDÈS, E. J., AND TAO, T. Near-optimal signal recovery from random projections: Universal encoding strategies. *Information Theory, IEEE Transactions on* 52, 12 (2006), 5406–5425.
- [7] CHAIKEN, R., JENKINS, B., LARSON, P.-R.A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. Scope: easy and efficient parallel processing of massive data sets. *PVLDB* 1, 2 (2008).
- [8] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.
- [9] CHAUDHURI, S. An overview of query optimization in relational systems. In *PODS* (1998), pp. 34–43.
- [10] CLEARY, J., AND WITTEN, I. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications* 32, 4 (1984), 396–402.
- [11] DEUTSCH, P. DEFLATE compressed data format specification version 1.3. <http://www.ietf.org/rfc/rfc1951.txt>, 1951.
- [12] DONOHO, D. L. Compressed sensing. *Information Theory, IEEE Transactions on* 52, 4 (2006), 1289–1306.
- [13] DUARTE, M. F., DAVENPORT, M. A., TAKHAR, D., LASKA, J. N., SUN, T., KELLY, K. F., AND BARANIUK, R. G. Single-pixel imaging via compressive sampling. *Signal Processing Magazine, IEEE* 25, 2 (2008), 83–91.
- [14] FANG, Y., CHEN, L., WU, J., AND HUANG, B. Gpu implementation of orthogonal matching pursuit for compressive sensing. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems* (Washington, DC, USA, 2011), ICPADS '11, IEEE Computer Society, pp. 1044–1047.
- [15] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 29–43.
- [16] ILYAS, I. F., BESKALES, G., AND SOLIMAN, M. A. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)* 40, 4 (2008), 11.
- [17] PATI, Y. C., REZAIIFAR, R., AND KRISHNAPRASAD, P. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Signals, Systems and Computers, 1993. 1993 Conference Record of The Twenty-Seventh Asilomar Conference on* (1993), IEEE, pp. 40–44.
- [18] TROPP, J. A., AND GILBERT, A. C. Signal recovery from random measurements via orthogonal matching pursuit. *Information Theory, IEEE Transactions on* 53, 12 (2007), 4655–4666.
- [19] WRIGHT, J., YANG, A. Y., GANESH, A., SASTRY, S. S., AND MA, Y. Robust face recognition via sparse representation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 31, 2 (2009), 210–227.
- [20] YING YAN, JIAXING ZHANG, B. H. J. M. Z. Z., AND MOSCIBRODA, T. Distributed outlier detection using compressive sensing. Tech. rep., Microsoft Research, 05 2014. <http://research.microsoft.com/apps/pubs/default.aspx?id=217534>.
- [21] ZHANG, Y., ROUGHAN, M., WILLINGER, W., AND QIU, L. Spatio-temporal compressive sensing and internet traffic matrices. In *ACM SIGCOMM Computer Communication Review* (2009), vol. 39, ACM, pp. 267–278.
- [22] ZHOU, J., BRUNO, N., WU, M.-C., LARSON, P.-R.A., CHAIKEN, R., AND SHAKIB, D. Scope: parallel databases meet mapreduce. *VLDB J.* 21, 5 (2012).
- [23] ZIV, J., AND LEMPEL, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536.