Leveraging High-Performance In-Memory Key-Value Data Stores to Accelerate Data Intensive Tasks

Nesrine Khouzami, Jordà Polo and David Carrera

Technical Report UPC-DAC-RR-CAP-2015-19

Department of Computer Architecture - Universitat Politècnica de Catalunya

Barcelona Supercomputing Center (BSC)

Abstract

Big Data is an invading term referring to huge and complex set of data increasing permanently to reach a size typically of the order of tera or exabytes. The massive amount of data is created from diverse range of applications. For several decades, most data storage were using relational database management systems (RDBMS), but for very large scale datasets RDBMSs are not suitable because joins and locks influence negatively their performance in distributed environments. This limitations gave birth to NoSQL data systems to offer properties that traditional databases simply cannot provide. The main goal of NoSQL databases targets two key criteria: fast accessibility either for reading or writing using a simple call level interface with no concurrency model to access the data; and scalability to be able to manage a massive amount of data. Key/Value stores are a particular case of NoSQL databases, and consist of data objects called values which are associated with distinct character strings called keys stored in form of records. In this work we explore the suitability of a scalable key/value store for integration with parallel programming models.

1 Introduction

Big Data [12] is an invading term referring to huge and complex set of data increasing permanently to reach a size typically of the order of tera or exabytes. The massive amount of data is created from diverse range of applications, such as high-frequency financial trades, social web applications, large network monitoring, etc.

In order to store this data, companies dedicate clusters with thousands of commodity hardware machines. To access the stored data, high performance and high availability are fundamental requirements for many companies. Indeed, massive read and write requests have to be processed without noticeable latency. Thus, Big Data technologies and architectures have to be wisely designed to efficiently create, retrieve and analyze a very large volume of a wide variety of data.

In fact, for several decades, most data storage were using relational database management systems, RDBMSs [16]. To query the database, SQL [16], the Structured Query Language, is used. SQL allows to construct powerful queries, full of joins, unions and complex operations over large table, to analyze and sub-sample structured data according to the needs. Using RDBMSs, multiple clients could access the data storage in a consistent way.

Most applications using relational databases require full ACID (Atomicity, Consistency, Isolation and Durability) support [10] to guarantee that transactions are processed reliably. These strict properties make RDBMSs an inappropriate fit for many applications expecting to take advantage of a high-performance, elastic, and distributed data environment.

Indeed, RDBMSs are not suitable in this domain because joins and locks influence negatively the performance in distributed systems. Relational databases are stretched to their limits in terms of scalability. In addition to high performance, several companies require that databases must be easily replicated so that data is always available in case of data center failures. In this context, replication techniques offered by RDBMSs are limited since they are basically relying on consistency instead of availability.

The aforementioned limitations of RDBMSs to deal with Big Data requirements gave birth to NoSQL data systems [18] to offer properties that traditional databases simply cannot provide. The main goal of NoSQL databases targets two key criteria: fast accessibility either for reading or writing using a simple call level interface with no concurrency model to access the data; and scalability to be able to manage a massive amount of data. As additional functionalities, NoSQL increases the flexibility for data manipulation as it provides a relaxed approach to model data. This gives the ability to dynamically add new attributes to data records.

NoSQL databases emerged and became prevalent in several companies including Google using BigTable [14], Facebook using Cassandra [1, 19], Amazon using Dynamo [15], etc. NoSQL databases can be categorized into 4 types: key-value

stores, document stores, column family stores and graph databases [18]. Our focus will concern key-value stores.

Indeed, a key-value store consist of data objects called values which are associated with distinct character strings called keys stored in form of records. Key-value stores are known as schema-less models as values content could be defined by the user. All operations to process the data are based on keys. Most key-value stores expose the basic API to put, get and remove data. Key-value stores tend to be easy to scale and have good performance to access data.

In conclusion, the motivations behind the development and usage of NoSQL databases can be summarized by the need for high scalability as well as processing efficiently a large amount of data distributed among many servers. Consequently, to take benefit from this features, key-value store notably can be smoothly integrated with existing programming models.

2 State of the Art

2.1 Key-Value Stores

Key-value stores are a type of NoSQL [18] database having a different structure from the traditional relational database designs. Key-value store structure is similar to maps or dictionaries where data is addressed by a unique key. The key is the only identifier with which we can retrieve data. There is no relationship between values, they are isolated and independent from each other. Most key-value stores are data schema free. Different kinds of data containers, data types and objects are used to accommodate this. Key-value stores are generally appropriate solutions for applications with only one kind of objects easy to look up based on one attribute. Although they present a restricted API, key-value stores offer a significant performance and scalability advantages compared to traditional databases.

Most key-value store systems try to satisfy the following features.

- Consistency:

  Most key-value store systems pretend to be "eventually consistent", which means that updates altering a specific data must be propagated over all nodes. The consistency differs between systems, those using replication have to make sure that all replicas are consistent, retrieving a value of given key should be always the same. To achieve this, those systems are providing mechanisms for some degree of consistency, such as multi-version concurrency control (MVCC) [11].

- Throughput:

  Throughput metric is considered one of the primary goals of database systems, it focuses on the number of transactions that can be completed in a given unit of time.

- Scalability:

  We say that a key-value store system satisfies the metric of scalability if it manages to distribute the data and the load of read/write operations over many servers which may increase the throughput of the system. Here we talk about "horizontal scalability". Some systems consider as well the "vertical scalability" where many CPUs share the RAM and disks. One important technique to ensure vertical scalability is to use the RAM instead of the disk to store the data.

- Availability:

  The availability feature means to guarantee that reads and writes always succeed. The data has to be in any particular time available for the user.

- Partition Tolerance:

  Data has to be stored in multiple locations. Through distribution, the system can handle failures of some individual servers that can't be reached and continue to operate as whole which guarantee that previous features are still maintained.

There are several implementations of key-value store available in market, they offer different functionalities and improvements over others. We examine some of them in this section.

- **Redis**

  Redis [2] uses a data structure which includes strings, lists, sets, sorted sets and hashes. Redis does not support complex queries, it implements insert, update, delete and lookup operations. All Redis commands are executed atomically. In Redis, the servers keep the dataset in RAM, but they write the entire data to disk at configurable intervals for backup and system shutdown. The dump is loaded every time the server restarts. The client side does the distributed hashing over servers. A cryptographic hash over the key is calculated and a server is chosen according to the result. A master-slave replication is available in Redis to make sure that even few lost records are not acceptable.

- **Voldemort**

  Voldemort [5] is an advanced key-value store used at LinkedIn by numerous services. Voldemort supports lists, JSON objects or other serialization in addition simple scalar values. The data is accessed through call of operation get(), put() and delete(). Hashing is used to distribute data around nodes. Each node is independent of others with no central point of failure. The system adapts automatically whenever nodes are added or removed from the database cluster and data partitioning is transparent. In Voldemort, data can be stored in memory but also plugging in storage engines (BDB-JE, MySQL, Read-Only) is permitted. Voldemort uses MVCC for data updates. It provides asynchronous replication which lead to a tunable consistency.

- **Cassandra**

  Cassandra [1, 19] is an advanced key-value store which was originally open sourced by Facebook in 2008. To represent data, Cassandra uses as data structure building blocks to form up to 4 or 5 dimensional hash. Basically, values can be a collection of other key-value pairs. The hierarchy of database starts with the keyspace that contains a column family storing a group of keys and values pairs. Unlike other key-value stores, a query for a ordered range of keys can be processed . Cassandra scales horizontally in the purest sense with provision of high availability service. Canssandra's architecture

follows a masterless ring design making it easy to setup and to maintain. It has no single point of failure, we can add new nodes to an existing cluster without shutting it down, which makes it offering continuous availability. For applications where Cassandra's eventual consistency model is not adequate, "quorum reads" of a majority of replicas provide a way to get the latest data.

- **Dynamo**

  Dynamo is Amazon's key-value store [15]. It has a simple interface with two operations get() and put(). Dynamo applies a hash on the key to generate an identifier which is used to determine the storage nodes responsible for serving the key. The hash maps the keys over a single Distributed Hash Table ring without requiring any centralized coordination. For Dynamo, availability is more important than consistency which is sacrificed for the ability to write at any time. Actually, all updates will be eventually done in all data nodes, making all data consistent, but there is no clue on how much time the consistency will take to be achieved . Dynamo system is designed to be highly scalable, it's using commodity hardware which allows an important number of datacenters. As Amazon is dealing with important data, it does replication on multiple servers at different locations. The concept of Service Level Agreement (SLA) is considered in Dynamo database which is essentially a contract between the client and the server in order to establish a time in which the data must be processed.

- **BerkeleyDB**

  BerkeleyDB project starts as a new implementation of various methods in DataBase Manager (dbm) system [20]. It provides different interfaces with C, C++, Java, etc. Records consist of key-value pairs that can be any data type or structure supported by the programming language. There are three different access methods that BerkeleyDB uses to operate on keys : B+tree, hashing and fixed-or-variable-length records (Recno). The keys and values size allowed in berkeleyDB doesn't exceed 4GB, and the size of the database reaches 256 petabytes. BerkeleyDB uses the host filesystem as backing store for the database, but it allows also for some applications that do not require persistent storage to create databases in main memory. For read-only use, BerkeleyDB can memory-map the database so that applications operate directly on records on pages. The size of the pages is configurable and users can define it according to their preferences to control system performance. BerkeleyDB supports cursors and joins, besides it guarantees the ACID properties. It uses checkpoint to indicate consistent data committed in the disk, here consistency is not totally guaranteed.

- **Tokyo**

Tokyo project [4] has Tokyo Cabinet as back-end server and Tokyo Tyrant as client library that provides outside access to the data stored in the server. Tokyo Cabinet is dealing with multiple data structures to organize the content. It supports for example hash indexes, B-tree and fixed-length array for both in-memory or disk storage. Tokyo project implements the common get/set/update operations. It claims the support of locking besides ACID transactions and more other complex operations to atomically update data. The project supports asynchronous replication, but recovery of a failed node is manual and there is no automatic sharding. Regarding the speed, Tokyo project is fast but only for small data sets (around 20 million rows) but it has a fairly good horizontal scalability.

- **Riak**

  Riak is a Dynamo-inspired key-value store [7]. It's written primarily in Erlang. The data is composed of a pair of key and value stored in a bucket considered as a namespace and gives the ability to a same key to exist in different buckets. Riak implements both a native interface and http API allowing the manipulation of data through the methods GET, PUT, POST and DELETE. The system supports MVCC and makes track of updates on objects through a vector clock to repair data that is out of synchronization. Riak is using consistent hashing to distribute data between the cluster nodes, it's architecture is simple and symmetric, all nodes are equal, no master node to track the system status. Riak also include MapReduce mechanism to share work between nodes. Replication is supported in Riak to guarantee that data is always safe whenever a node goes down. Riak is able to rebalance data if nodes join or leave the cluster. Besides, Riak claims to scale linearly.

2.2 Scalable Key-Value (SKV)

To explore key-value database systems, our work adopts the Scalable Key-Value (SKV) store project [3].

SKV is a key-value store developed by IBM to exploit massively parallel, processor-in-memory environments such as Blue Gene (BG) machines. It was designed for the Blue Gene Active Storage project [6] which explores the integration of non-volatile memory and Remote Direct Memory Access (RDMA) [22] networks and benefit from such an approach.

The design of SKV is BerkeleyDB-inspired. Besides, SKV is using the communication paradigm RDMA. SKV presents a parallel client and parallel server. The storage of data is done in memory, each node providing storage runs a parallel SKV server. All server nodes form SKV Server group. SKV client library runs in an MPI parallel environment providing an API managing the communication between the client and the server. The SKV clients and servers operate on a

request-response model. Any valid SKV command can be sent to any server. SKV server group should run on homogeneous parallel machines, each node run exactly 1 server. The SKV client and the SKV server can run in different or same hosts.

SKV library provides the same standard C/C++ API as BerkelyDB. But data is not stored in a file, SKV library uses a messaging interface to exchange data with the SKV server group. The SKV client-server protocol involves short control messages handled in an RPC style exchange and bulk transfers handled via an RDMA type exchange. Client connection to the SKV server group is managed so that a single connection point provides RDMA access to all servers across the machine.

SKV uses as data structure a Partitioned Data Set (PDS) to store tables. a PDS is a distributed container of key-value records. When a PDS is opened, a data structure is returned which may be shared with other processes via message passing to enable a parallel process to rapidly gain global access.

The SKV server that creates a PDS is the PDS root node. Each SKV server uses a unique ID to name the created PDS. The unique ID is formed in such a way that the SKV server address is an accessible component of the ID. It is up to the creator of the PDS to keep track of the PDS ID including any mapping from a name space map.

A PDS is read and written on a record basis. To access a record, a key must be provided. Parts of records may be read or written. Records may be compressed. Currently, records are distributed to nodes based on a hash of the key, then stored in a red-black tree in memory. Every key hashes to exactly 1 node. A PDS corresponds to a database table and a record corresponds to a database row.

SKV client has the ability to request the server group to take a current snapshot of the data image and save it to persistent storage on disk at a specified location. The server group could then be restarted with the data from a snapshot image.

Figure 1 illustrates the SKV overall functioning of a SKV deployment. A full description of the SKV API has been included in Appendix **??**.

2.3 Data management in shared memory environment

Our target environment to emulate key-value store integrated with programming model will be Distributed Shared Memory (DSM) [21]. Indeed, a distributed shared memory is a mechanism allowing end-users' processes to access shared data without using inter-process communications.

In other words, the goal of a DSM system is to make inter-process communications transparent to end-users. In any case, implementing a DSM system implies to address problems of data location, data access, sharing and locking of data, data coherence. Such problems are not specific to parallelism but have connections with distributed or replicated databases management systems. Several programming models implement features to manage the access to distributed regions of

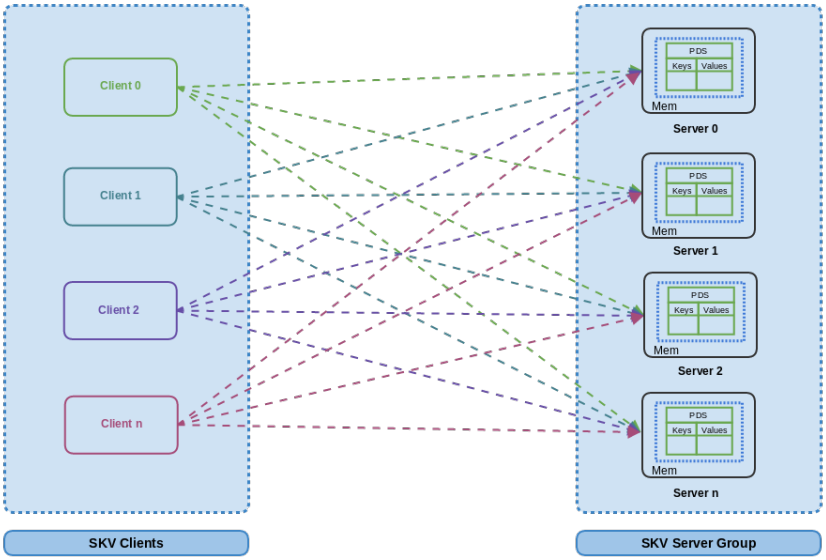

Fig. 1: SKV Overview

data.

As examples, we mention StarPu [9] which is a programming model offering an infrastructure having as base architecture accelerators and implementing features to partition the application data, but in an explicit way as the user has to call an API to deal with data.

There is also Sequoia [17] alternative which tries to map application with appropriate engines taking into consideration the memory distribution. Sequoia compiler supports complex data specifications, it's up to the user, at compiling time, to specify the desired architecture so that the compiler generates the right code to deal with data transfers.

Mint [23] is another project targeting GPU platform able to generate optimized code to access data located in different regions and transfer them from host memory to GPU memory. This is applicable in clusters environment as well.

Another interesting work targets OmpSs programming model [13], it presents a mechanism handling automatically regions of data that can be overlapped on architectures with disjoint address spaces. In this work, a data structure and mechanisms were designed and implemented in order to specify which data must be transferred between distributed address spaces. OmpSs offers different clauses that should contain variables representing the data that will be managed.

3 Results

The main goal of this work is to perform a preliminary study and select an efficient key-value store that can be integrated with advanced programming models to play the role of data backend for many applications providing a considerable amount of memory.

This section describes the different benchmarking efforts carried on so far to evaluate the performance and scalability of state-of-the-art high-performance key-value stores.

The key-value store initially chosen for the evaluation is SKV (for further details, please refer to Section 2.2). It is a Key-Value store specifically designed for high-performance environments, leveraging RDMA to deliver high-throughput, low-latency and low-resource consumption across workloads.

3.1 Data Distribution Strategies: Dependent vs Independent servers

Most Key-Value stores rely on well-studied data distribution strategies. In most cases, universal hash functions are leveraged to ensure that keys (and their corresponding data) are evenly distributed across the back-end servers, independently of the data nature. SKV is no different in this sense, and it brings a built-in universal hash function that in most cases results in a good data distribution mechanism across SKV servers.

When looking at high-performance KV solutions as SKV though, it is worth noting that beyond a proper data distribution function, it is also important too look at the cost of performaning a well-balanced data distribution. In SKV, servers are usually deployed as a set of MPI processes that now each other, and they can perform data-rebalancing strategies as needed.

This fact, that can be seen as a positive point of the KV store, can potentially bring also some limitations and bottlenecks on the server side, if excessive coordination between the servers is required. To evaluate the cost of such operations, several of our experiments were run in what we called *Independent Servers* mode, in which we deploy several independent servers that don not know each other, and push the data distribution responsibility to each one of the client processesm, that will have to select the proper server in which data is stored for every KV operation that is performed. On the other hand, the so called *Dependent Servers* mode is the regular SKV mode in which the built-in hash function is used, and all servers know each other.

The full development of the Independent Servers mode is part of this thesis and therefore is evaluated and discussed in the following subsections.

3.2 Execution Environment

All our experiments have been executed in BSC supercomputer, Marenostrum-3 [8]. For all scenarios, we were making synchronous insertion of 10M records, we insert integers (*uint64_t*), 8 Bytes as size of keys and 8 bytes for values as well. SKV clients and servers have been running in disjoint nodes. The number of clients and servers ranges from 1, 2, 4, 8, 16 up to 32 nodes.

3.3 Dependent Servers

In this scenario, we run one server group with N number of SKV server processing in separate nodes. We did the insertion of records using sockets and RDMA verbs.

3.3.1 Using sockets

Both figures 2 and 3 show the elapsed time and the speedup to insert 10M records. For 1 node, the execution time takes around 370 seconds. With 2 nodes, we have as average time about 230 seconds, with 4 nodes, the elapsed time is equal to 67 seconds. For 16 and 32 nodes, we have respectively 32 and 22 seconds as insertion time. Here, we see that SKV is scaling, but we are not reaching full scalability.

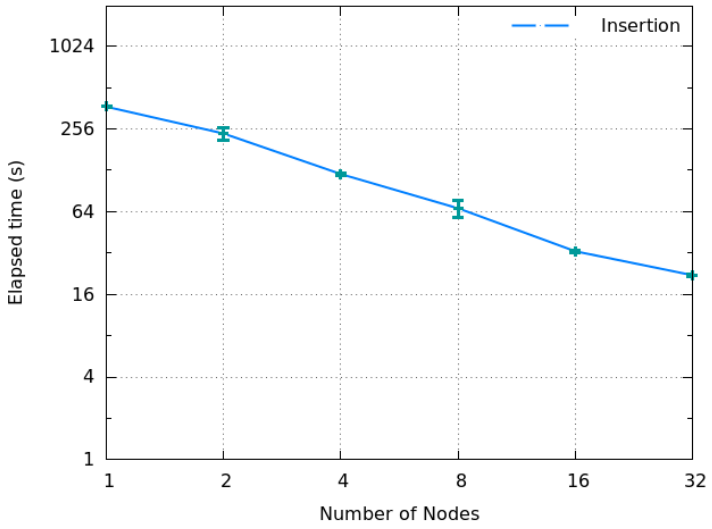

Fig. 2: Elapsed time of insertion

3.3.2 Using verbs

Using RDMA verbs, we have better execution time comparing to sockets, as shown in figure 4. Actually, for 1 node, the insertion of 10M records took 162 seconds,

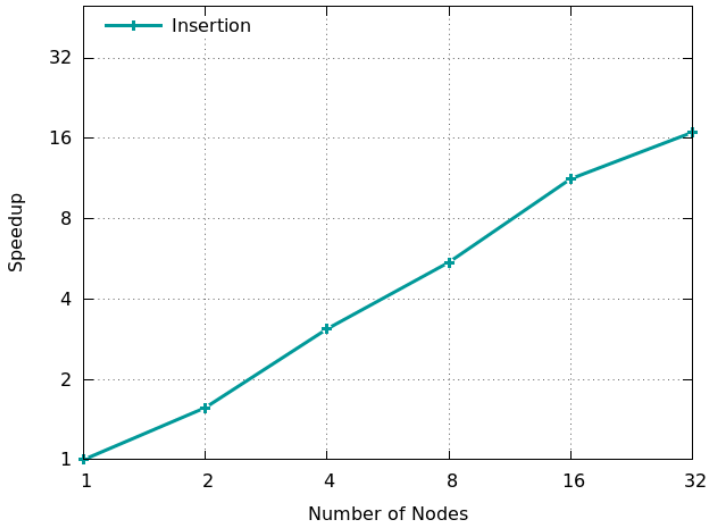

Fig. 3: Speedup of insertion

for 2 nodes, it took a bit less, 110 seconds. With 4 and 8 nodes, the execution time is getting better, we have respectively 58 and 48 seconds. Then the insertion took about 22 and 18 seconds using respectively 16 and 32 nodes. Figure 5 shows that SKV didn't reach linear scalability yet. But we notice that with verbs the insertion is faster than with sockets thanks to the low-latency that RDMA offers.

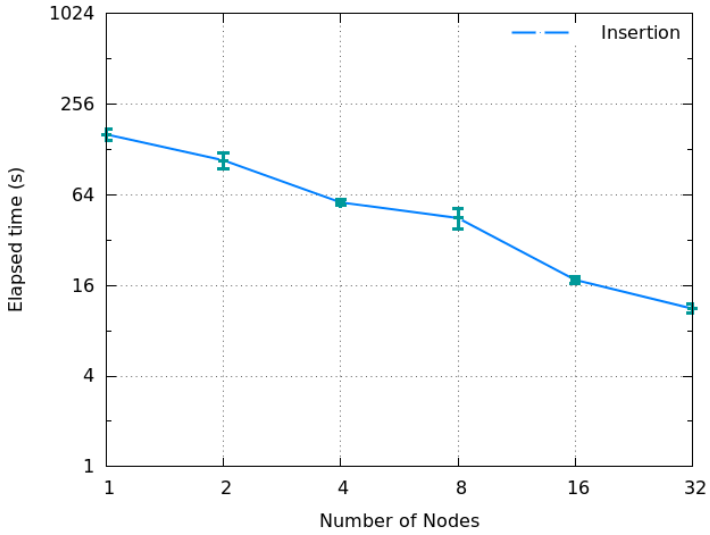

Fig. 4: Elapsed time of insertion

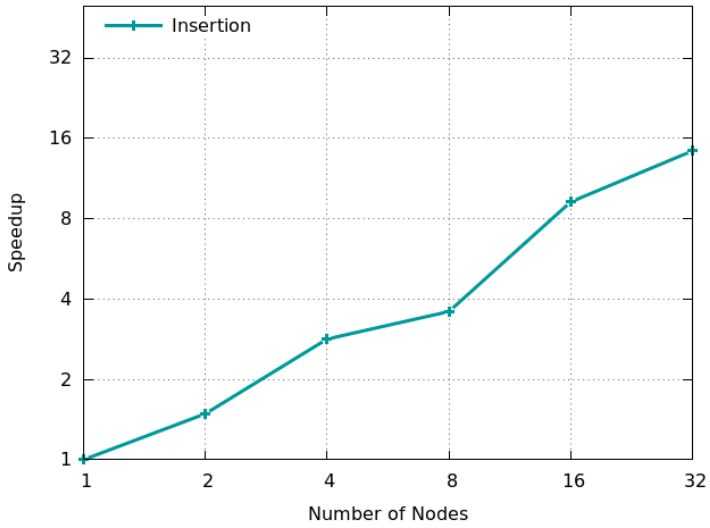

Fig. 5: Speedup of insertion

3.4 Independent Servers

As second configuration, we run N single servers in separate nodes. The servers do not belong to one server group.

3.4.1 Using sockets

Figures 6 and 7 show the insertion time and speedup using sockets. Comparing to dependent servers, the insertion took less time with 1 server up to 16. With 32 nodes, the execution time is surprisingly increasing comparing to 16 nodes.

3.4.2 Using verbs

Using verbs, figures 8 and 9 show that the execution time is slightly better with one and 4 nodes than execution time of insertion sockets-based. The benefit of using verbs appears with more nodes, with 16 and 32 nodes, the execution time took 17 and 15 seconds comparing to 20 and 24 respectively with sockets. We notice in this scenario linear scalability up to 8 nodes. Then, as well as for dependent servers, SKV is not reaching full scalability with synchronous insertion. Next scenarios will consider the asynchronous insertion to see the influence on scalability.

3.5 Conclusions

The main conclusions of the study show that although SKV is not yet scaling linearly in MareNostrum-3, it is still a well-performing KV store for Infiniband-equipped cluters. The RDMA implementation is faster than regular sockets im-

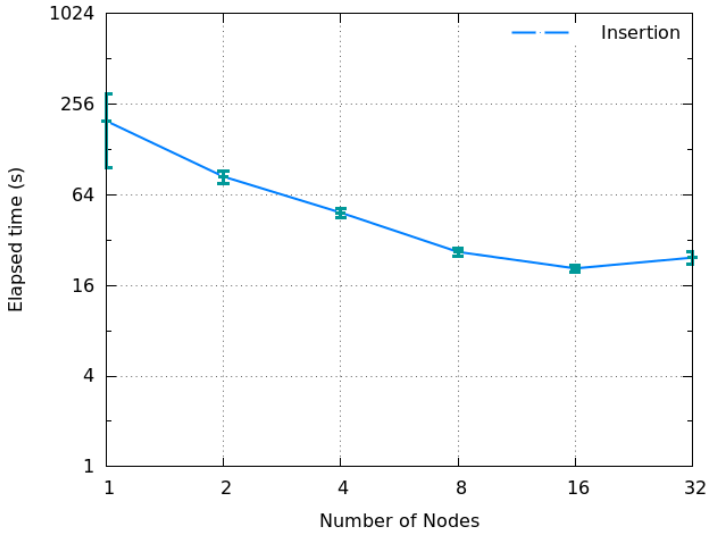

Fig. 6: Elapsed time of insertion

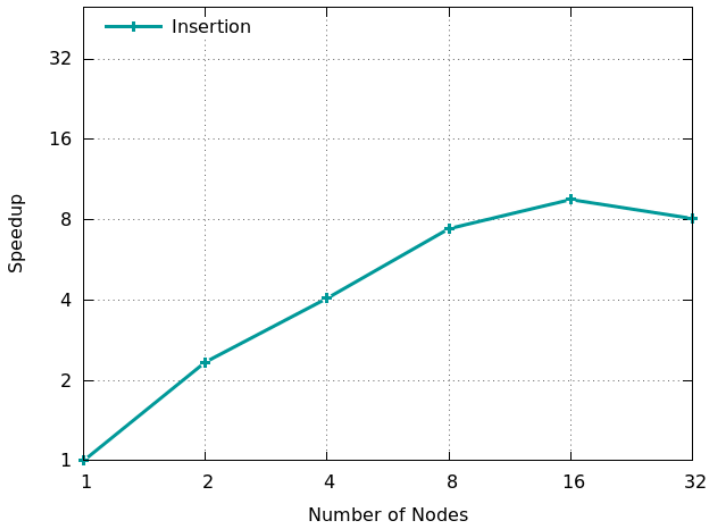

Fig. 7: Speedup of insertion

plementation (comparable to other KeyValue stores not designed for High Performance environments). It not justs performs better, but it also results in lower resource consumption levels of the cluster nodes, as most of the computation is off-loaded to the Infiniband HBA cards.

Regarding Dependent vs Independent Servers, we can confirm that the overheads that may have been introduced by the original Dependent Servers model are negligible, not degrading the performance nor the scalability properties of

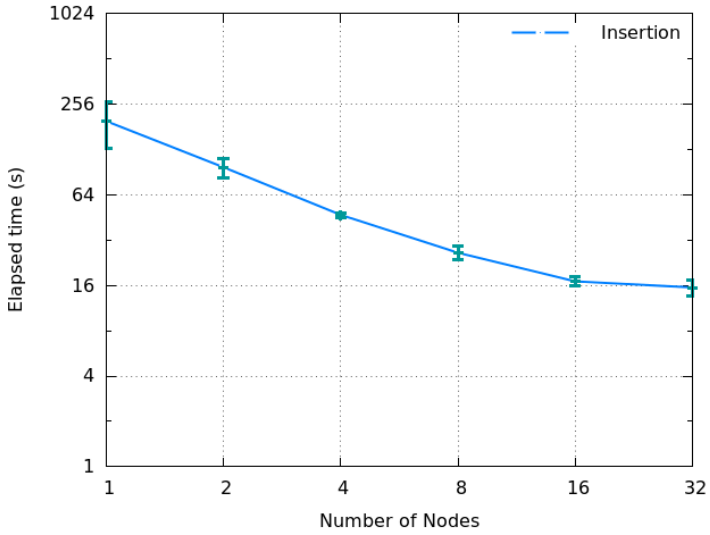

Fig. 8: Elapsed time of insertion

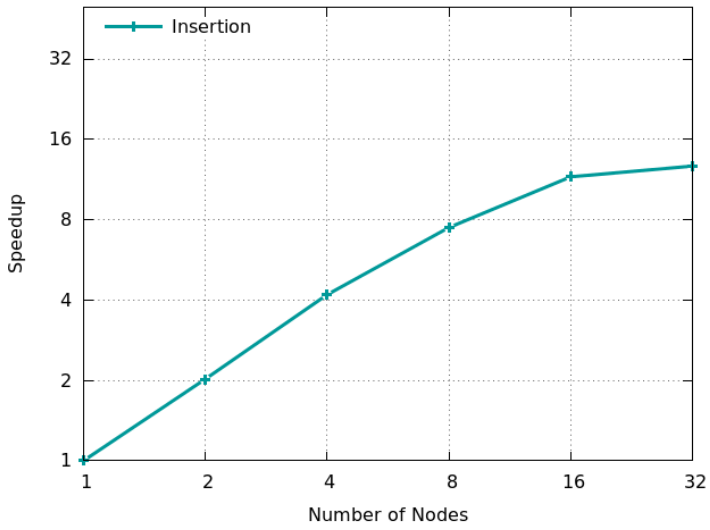

Fig. 9: Speedup of insertion

SKV when comparing to the Independent Servers mode.

Beyond key insertion tests, other experiments are being conducted that have not been included here, involving retrieve operations, iterators and asynchronous operations. Most of these experiments are still running or under evaluation .

In summary, the version that will be used for the experimental integration with the parallel programming models will be the verbs SKV implementation, with Dependent servers and quite possibly based on asynchronous operations based on

our initial findings.

4 Acknowledgments

This project is supported by the IBM/BSC Technology Center for Supercomputing collaboration agreement. It has also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 639595). It is also partially supported by the Ministry of Economy of Spain under contracts TIN2012-34557 and 2014SGR1051 and by the BSC-CNS Severo Ochoa program (SEV-2011-00067).

Bibliography

[1] Cassandra. http://cassandra.apache.org/, 24-06-2015.

[2] Redis. http://redis.io/, 24-06-2015.

[3] SKV. https://github.com/scalable-key-value/code, 24-06-2015.

[4] Tokyo. http://fallabs.com/tokyocabinet/, 24-06-2015.

[5] Voldemort. http://www.project-voldemort.com, 24-06-2015.

[6] Blue gene active storage. https://institute.lanl.gov/hec-fsio/workshops/2010/presentations/day1/Fitch-HECFSIO-2010-BlueGeneActiveStorage.pdf, 26-06-2015.

[7] Riak. http://basho.com/products/#riak, 26-06-2015.

[8] Marenostrum. http://www.bsc.es/support/MareNostrum3-ug.pdf, 27-06-2015.

[9] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[10] P. Bernstein and E. Newcomer. Atomicity, consistency, isolation and durability. *Principles of Transaction Processing*, pages 9–15.

[11] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[12] M. A. Beyer and D. Laney. The importance of 'big data': a definition. *Stamford, CT: Gartner*, 2012.

[13] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Implementing ompss support for regions of data in architectures with multiple address spaces. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 359–368. ACM, 2013.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007. ISSN 0163-5980.

[16] R. Elmasri. *Fundamentals of database systems*. Pearson Education India, 2008.

[17] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[18] J. Han, E. Haihong, G. Le, and J. Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[19] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[20] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[21] J. Protic, M. Tomasevic, and V. Milutinović. *Distributed shared memory: Concepts and systems*, volume 21. John Wiley & Sons, 1998.

[22] A. Romanow and S. Bailey. An overview of rdma over ip. In *Proceedings of the First International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2003)*, 2003.

[23] D. Unat, X. Cai, and S. B. Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.