2017 IEEE 33rd International Conference on Data Engineering

Making Big Data Simple with dashDB Local

Sam Lightstone
IBM
Toronto, Canada
e-mail: light@ca.ibm.com

Russ Ohanian
IBM
Marlborough, USA
e-mail: rohanian@us.ibm.com

Michael Haide
IBM
Boeblingen, Germany
e-mail: HDE@de.ibm.com

James Cho
IBM
Portland, USA
e-mail: jamescho@us.ibm.com

Michael Springgay
IBM
Toronto, Canada
e-mail: springga@ca.ibm.com

Torsten Steinbach
IBM
Boeblingen, Germany
e-mail: torsten@de.ibm.com

***Abstract*—In this paper we introduce dashDB Local, a new Big Data & SQL warehousing technology from IBM designed to provide dramatic simplification over traditional data warehousing deployments, while offering next generation query performance and the analytic richness of the Apache Spark ecosystem. Designed as a Software Defined Technology, dashDB Local automatically adapts to hardware platforms to provide simplified deployments. In experiments one multiple workloads, we have achieved deployment times for large clusters in < 30 minutes while providing dramatic workload performance speedups of several factors on multiple workloads under study compared to industry leading appliance technology and market leading data warehouses. We see dashDB Local providing five novel value propositions: 1. Improved deployment with automatic adaptation to the target hardware enabling fast deployment in minutes of fully configured data warehouse and Big Data clusters from gigabytes to petabytes. 2. Superior workload performance through advances in in-memory columnar algorithms. 3. Rich polyglot language support, supporting many dialects of Big Data, SQL languages. 4. Full stack integration of Apache Spark into the processing engine, co-existing with the rich SQL engine. 5. Compatibility with dashDB as a service in the cloud, providing on-premises and cloud flexibility to develop and run applications and analytics where desired.**

Keyworkds - Big Data, Data Warehousing, Columnar, SQL, Query, Apache Spark, Docker, Linux Container.

I. INTRODUCTION

For the past 30 years Data Warehouse technology has dominated the space of data analytics, with MPP clusters that scale to massive compute and storage. The complexity of these systems eventually gave rise to Data Warehouse Appliances that provided pre-configured deployment blocks. The rise of open source components such as Hadoop, Hive, Spark maintained the cluster computing approach to data analytics, but provided open source components and attractive programming paradigms such as schema on read. However, what the Hadoop based Big Data technologies offered in feature and flexibility they lacked in simplicity, and composing robust solutions from these collections of open source has proven challenging for many companies. Moreover, the performance of HDFS has been a limiting factor in achieving high performance analytics, giving rise to the advent of Apache Spark for in-memory analytics for scenarios where RAM is plentiful. Yet, the full opportunity for high performance in-memory optimized analytics remains the domain of Data Warehousing, with columnar vector processing engines dominating the space. In-memory performance of these systems are often several times faster than current performance of Apache Spark, which itself is several times faster than performance over raw HDFS. With the introduction of dashDB Local[1] we provide a flexible platform with five significant novel technology improvements:

1. Software Deployment and update Simplicity. Improved deployment based on Linux container technology, with automatic adaptation to the target hardware. This enables fast deployment of fully configured Big Data clusters from gigabytes to petabytes. This strategy allows user to deploy fully configured clusters of dashDB Local in minutes. This also allows update of entire complex software stacks by simple container replacement and restart.
2. Query Performance. Fast CPU-optimized workload performance based on the next generation of IBM's BLU Acceleration technology[11][21], often performing workloads several times faster than leading warehouse appliances, cloud warehouse and many column stores.
3. Rich polyglot language support. dashDB supports many dialects of Big Data languages including multiple SQL language variants (Oracle, DB2, PostgreSQL, Netezza), PL/SQL and SQL PL,
4. Apache Spark. First of a kind integration of Apache Spark into the core analytics engine. This enables dashDB Local to support Spark SQL, PySpark, and Apache Spark packages such as SparkR, MLLib for machine learning and interactive Notebooks.
5. Mutually available as a software defined on-premises deployment and as a cloud service.

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.164

dashDB Local uses the same query engine as IBM's dashDB cloud service, which guarantees that application code written on either of these will execute on the other. This allows development organizations in need of an on-premises solution such as dashDB Local several advantages:

- Projects can be prototyped quickly in the cloud on dashDB as a service, and then hardened into a production version on-premises with dashDB Local as they mature.
- Dev & QA. As many development and QA operations (certainly not all) can be performed on synthetic data and often on small volumes of data, the public cloud offers a flexible and low cost platform for fast provisioning of systems where development and QA can be performed before promoting code changes to the production system on-premises.
- Cloud as hot backup. Many companies prefer to keep their production systems on-premises, but are increasingly cloning the data in these systems to the cloud for use as a hot standby for Disaster Recovery purposes. This paradigm requires the availability of a semantically similar analytic engine in the cloud, such as dashDB.

In the sections that follow we will describe the implementation that supports these value propositions, provide some recent measurements on query performance relative to existing commercial systems, and some thoughts on future work.

II. IMPLEMENTATION

A. *Deployment with Linux Container technology*

dashDB Local combined two principles to achieve deployment simplification. The first is adoption of Linux containers. By packaging the dashDB software stack into a Docker container[2], provisioning dashDB Local can be as simple as one *docker run* command on a Linux server that has the Docker engine installed. Similarly, publically available Docker tooling, such as Kitematic, can be used to provide a user friendly graphical interface providing access to Docker hub search for "dashDB" followed by a single click on "CREATE" on Windows or Mac machines. Software stack updates use the same *docker run* command mechanism against a new version of the container and preserves the existing installation. To achieve this dashDB Local is available as a Docker container on a Docker Hub private repository accessible by registration. dashDB Local can be deployed on any supported Docker installation on a public or private Cloud, bare metal, or VM with minimal prerequisites. Entry-level hardware requirements start at 8GB RAM and 20GB of storage, that is suitable for a development / test environment or QA work on your laptop. Larger servers such as Xeon e7 4 x 18 core 72 way machines with 6 TB RAM servers can also be used or a cluster of them together for very large data volumes. Persistent durable storage of your choice must be mounted in */mnt/clusterfs* to hold your data. dashDB has been designed to be flexible to use hardware that you already have in your data center or other infrastructure that is available.

However, a simple download and deploy process while helpful, is insufficient to achieve complete simplification. Big Data systems that run over complex hardware topologies have many elements of configuration, for the allocation of memory to functional purposes (caching, sorting, hashing, locking, logging, etc.), query parallelism degree, workload management infrastructure, and many others. To extend a simple "download" to a more complete simplified deployment dashDB Local include an automatic configuration component that detects several characteristics of the hardware environment, and adapts its configuration to optimize for the resources available. This includes automatic detection of CPU and core counts, and automatic detection of RAM. Combining the simplified deployment from Docker with the automatic configuration to hardware target system, we find dashDB is consistently able to deploy to large clusters in under 30 minutes, fully configured and instantiated, with workload management, memory cache, query optimization levels and parallelism configured to match.

The docker container packages all dependencies and requirements of the dashDB software stack (see Figure 1). A clear boundary and separation of responsibilities and functionality can be maintained. Existing monitoring and security overlays can be deployed according to data center standards on the host system outside of the container. The isolation of the dashDB local container keeps it independent and compatible with variety of operational requirements.

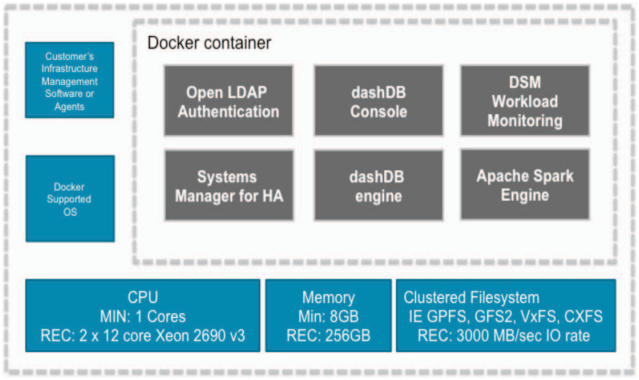

Figure 1 dashDB Docker Image

On the host, maintained by the user:

- Customer defines/owns hardware, Host OS, and the Cluster file system

- Customer is responsible for its own security updates on host OS
- Customer is responsible for managing the Docker engine
- IBM only requires Docker client running, POSIX compliant clustered file system for MPP
- Customer can create/start/stop/restart dashDB Local container(s)
- dashDB Local does nothing outside its container

Inside the Docker image provided by IBM:

- IBM defines dashDB Local software stack inside the Docker container, the application container is consistent and "stateless" from any changes
- New container images will be made available for stack updates via stop-and-rename of current container, and spinning a new container from new image (seconds to start container from new image, few minutes to start dashDB engine on large memory configurations)
- Only one dashDB Local container per Docker host

B. SQL Query Engine

The dashDB query engine for SQL operations is based on the BLU Acceleration technology from IBM. Details on this technology have been published describing its core processing strategies around columnar data organization, operations on compressed data, CPU-cache efficient algorithms, and CPU optimizations with SIMD instruction [9][10][11][12][17][21]. The advantages of columnar processing for Big Data analytics are extensively described in the literature [1][4][7] as are the benefits of memory-efficient processing over Big Data for analytics [7][8]. The BLU Acceleration technology has been deeply integrated in dashDB Local so that it is completely transparent. No configuration adjustments or system tuning are required by the user. A brief summary of the technology is provided here, and the reader is invited to refer to the referenced literature for a more in-depth treatment.

Broadly the storage and query performance benefits of this unique SQL engine come from seven architectural approaches:

1. Compression methods
2. Operating on compressed data
3. Columnar data representation
4. Data skipping
5. In-memory caching strategies
6. SIMD processing
7. Cache efficient algorithms for scan, join, grouping and aggregation

1) Compression methods

dashDB compresses data using a number of techniques. These include *minus encoding* methods for high cardinality numeric, and variations of Huffman encoding for lower cardinality fields known as *frequency encoding*. The use of frequency encoding methods ensures that data with the highest frequency of occurrence are encoded with the shortest representation. This allows dashDB to compress data as small as one bit, and in special circumstances even smaller. *Prefix compression* methods are also used to eliminate storage for commonly occurring string prefixes. Compression is then optimized globally per column as well as locally per storage page (page level). These techniques in combination have allowed dashDB to regularly compress data 2-3x smaller [15] than previous generations of compression techniques used in IBM products.

2) Operating on compressed data

The compression methods described above are not directly applied with an exclusive aim of optimizing compression for storage savings. A key idea in this technology is to perform SQL operations on data while they are in compressed form. To achieve this dashDB uses order preserving codes, meaning that the techniques above are applied in a strict manner so that within any frequency partition values are binary wise comparable for equality and inequality.

3) Columnar data representation

dashDB stores data on disk and in memory caches in columnar format. This means the within any storage page only values of a single table column are represented. For most analytic operations on Big Data this proves to be a huge efficiency, as only active columns of interest to the workload need to be fetched from disk. The combination of operating on compressed data, and limited caching to the columns of active interest in the workload creates a multiplicative effect of increasing the density of useful information in RAM cache.

4) Data skipping

Commonly analytic queries over Big Data have restrictive predicates. The most ubiquitous are predicates over a restrictive date fields. For example, a data repository may store data for seven years, but most queries ask questions over the most recent few months. To leverage this common phenomenon dashDB uses a data skipping technique whereby metadata is collected and stored on every column for (approximately) 1K tuples. The meta data itself is stored in the same columnar compressed representation that the user data is stored in, so that the same processing efficiencies can be obtained. However, since meta data is stored only every 1K tuples, the metadata is generally three orders of magnitude smaller than the user data. It can be scanned three orders of magnitude faster while consuming a concomitant small footprint in memory and on disk.

5) In-memory caching strategies

Many Big Data and RDBMs systems over the past 30 years have used in-memory caching to reduce the need for I/O. Most of these algorithms have been based on variations of LRU to form a victim replacement algorithm when data is larger than RAM cache size. For large data sets frequently accessed in Big Data the LRU strategies perform poorly. The problem with caching Big Data is that the low selectivity patterns of Big Data lend themselves to scanning strategies for data analysis over fast random access lookup structures (such as via indexing). Scanning strategies are almost mutually exclusive with LRU strategies, since the least recently accessed data at the end of a scan is the data that was at the top of the scan, meaning the top of the scan is rarely in RAM at the start of the next scan. To resolve this incompatibility we developed a cache replacement algorithm that maintained a notion of access frequency, but was less sensitive to the position of data in the table (near the top or the bottom). A novel probabilistic algorithm for buffer pool replacement determines which pages to victimize, ensuring that only hot pages of hot columns are retained in memory, maximizing the buffer pool's hit ratio. The algorithm was published in [13] and found to produce cache efficiency rates for Big Data style scanning within a few percentiles of optimal.

6) SIMD processing

The dashDB compression strategies described above employ several encoding schemes to reduce the values to mere bits, and packs them bit-aligned within words. Because of this multiple values for a column can usually be packed into a single word, achieving excellent compression, and can be processed together. It is not uncommon for tens of values to be packed into a single word. Single-instruction, multiple data (SIMD) instructions are now common in modern processors, but are restricted to power-of-2, byte-sized units. The BLU Acceleration technology in dashDB enhances these SIMD instructions with novel software-SIMD algorithms to apply predicates simultaneously on all values in a word, for any code size. This SIMD parallelism is additional to the parallelism achieved by scheduling strides of data to multiple threads running on multiple cores of each CPU. Since analytics queries common in Big Data workloads are generally *low selectivity*, and rarely touch only a few rows, the runtime always scans the data. This scan centric processing model makes the benefits of SIMD based scanning all the more important. It also largely eliminates the need for indexes other than those enforcing uniqueness are necessary or even allowed.

7) Cache efficient algorithms for scan, join, grouping and aggregation

Considerable literature has been published on the impact of memory latency on Big Data analytics. In many CPU bound systems a large percentage of the processing time is actually consumed by memory latency to RAM access [14]. For this reason the philosophy behind BLU Acceleration technology used in dashDB is that DRAM is too slow! Instead, all of the query algorithms aim to keep data in the processor's L3 or L2 caches where access latencies are much smaller. This is done by working on batches of rows called *strides*, and by partitioning data into L3 or L2 chunks for performing joins and grouping, as pioneered in Hybrid Hash Join [7] and MonetDB [8]. Since analytics queries common in Big Data workloads rarely touch only a few rows, the query runtime in dashDB always scans the data, and so no indexes other than those enforcing uniqueness are necessary or even allowed.

The result of this careful engineering is remarkable speed, compression, and simplicity. Entire workloads run on column-organized tables in dashDB are typically 10 to 50 times faster than the same workloads run on row-organized tables with secondary indexing[11][15].

All of the 7 techniques are combined into a shared nothing MPP scale-out architecture as shown in Figure 2, which scales to massive data and compute, supporting thousands of cores over petabytes of data.

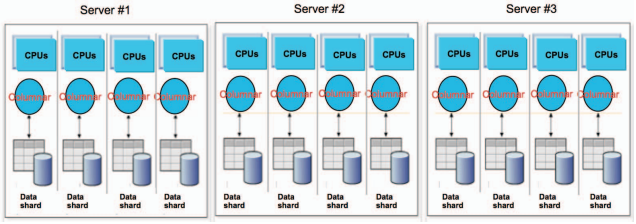

Figure 2 MPP shared nothing scale-out for dashDB

C. Polyglot language support

Part of the approach for dashDB is the idea that Big Data and Big Data applications come from everywhere. There is now a long history of several decades of application development for analytics, and many applications have been developed over this period using language extensions in Oracle, DB2, SQL Server, Netezza, Teradata, PostgreSQL, Spark, HiveQL, R, Python, Scala and many others. To achieve the broadest possible adoption we hope to simplify the process of moving applications to dashDB by providing native language support for the most important application domains and their language variants.

1) SQL language variants

We began with an ANSI standard compliant SQL compiler, and added extension for Oracle, PostreSQL, Netezza, and DB2.

a) Oracle specific language extensions

DML Extensions

- Outer Join Syntax using the (+)
- Anonymous Blocks - Execute adhoc PL/SQL scripts

- NEXTVAL and CURRVAL
- Hierarchical Queries (CONNECT BY)

- Scalar Functions: SUBSTR2, SUBSTR4, SUBSTRB, NVL, NLV2, INSTR, LPAD, RPAD, INITCAP, HEXTORAW, RAWTOHEX, LEAST, GREATEST, DECODE, TO_CHAR, TO_DATE, TO_NUMBER,
- Aggregation Functions: PRECENTILE_DISC, PRECENTILE_CONT, CUME_DIST, MEDIAN, VAR_POP, COVAR_POP, STDDEV_POP

Data Types
- VARCHAR2 (including empty string (null) and non-
- padded comparison semantics on varying strings)
- NUMBER
- DATE

DDL / Object Extensions
- ROWNUM
- DUAL table
- TRUNCATE TABLE
- CREATE GLOBAL TEMPORARY TABLE

b) Netezza / PostgreSQL specific SQL extensions

Data Types
- BOOLEAN

DML Extensions:
- JOIN USING
- LIMIT OFFSET
- expression::type casting
- ISNULL / NOTNULL
- ISTRUE/ISFALSE
- OVERLAPS OPERATOR
- ORDER BY ordinal
- GROUP BY output column name

Scalar functions:
- NOW , DATE_PART, POW, HASH, HASH4, HASH8, BTRIM, TO_HEX, bit operations: intNand, intNor intNnor, intNnot, STRLFT, STRRIGHT, STRPOS, AGE, NEXT_MONTH, DAYS_BETWEEN, HOURS_BETWEEN, SECONDS_BETWEEN, WEEKS_BETWEEN,
- Aggregation Functions: COVAR_POP, COVAR_SAMP, STDDEV_SAMP, STDDEV_POP

DDL Extensions:
- CREATE TEMP TABLE
- Data Types:
- INT2
- INT4
- INT8
- FLOAT4
- FLOAT8
- BPCHAR (only as cast target)

c) DB2 specific SQL language extensions

DML Extensions:
- VALUES clause
- NEXT VALUE and PREVIOUS VALUE

Scalar Functions:
- NORMALIZE_DECFLOAT,
- COMPARE_DECFLOAT,

Aggregation functions:
- COVARIANCE, COVARIANCE_SAMP, VARIANCE, STDDEV

DDL Extensions:
- DECLARE GLOBAL TEMP TABLE
- CREATE ALIAS
- Compound SQL (inlined) statements
- STATIC SQL

Data Types:
- DECFLOAT
- GRAPHIC

2) Semantic incompatibilities

While language compatibility includes many cases of creating a superset of the language elements (for example, the union of popular scalar functions used across products and services) there are also instances where the same or nearly similar syntax should behave differently in the context of one language variant versus another. Within dashDB multiple techniques are employed to deal with these colliding syntaxes. In the most extreme case a separate deployment images is used to provide a specific language semantic. This technique for example is employed for Oracle data type compatibility amongst others. The difference in comparison semantics associated with VARCHAR2 or Oracle's DATE format require a consistent definition across all entities within a given database. Were the dialect differences can co-exist a session variable is leveraged allowing individual sessions to decide the dialect to use when compiling SQL. The current session setting is stored with SQL objects created in a session such as views so that on subsequent reference

they adhere to the dialect as specified at creation time regardless of the accessing sessions setting.

3) Additional language interfaces

In addition to SQL language variants for in-memory data warehouse for the cloud there are numerous language interfaces that remain important for a wide range of applications that are also a focus of work for dashDB. dashDB provides the following application interfaces:

- ODBC
- JDBC
- .NET (C#)
- ADO
- OLEDB
- Visual Basic
- PHP
- Ruby
- Perl
- Python
- Node.js
- Embedded SQL
- PL/SQL for Oracle and Netezza variants
-

4) Netezza compatibility for in-database analytics

dashDB was designed with attention to the compatibility needs of Netezza users. A commonly used set of capabilities in this space is the collection of in-database analytics built into the Netezza appliance for R and numerous built-in algorithms. Drawing from this heritage of in-database analytics, dashDB has developed both R and Python analytics as well as commonly used machine learning algorithms. Figure 3 illustrates dashDB's integration with R Studio. dashDB provides R and Python language APIs to seamlessly delegate the heavy lifting of analytic computations to be performed with built-in database operations, be it with SQL statements or using built-in machine learning routines. Furthermore it allows users and application developers to extend the set of built-in functions with custom ones using the user defined extension (UDX) language framework.

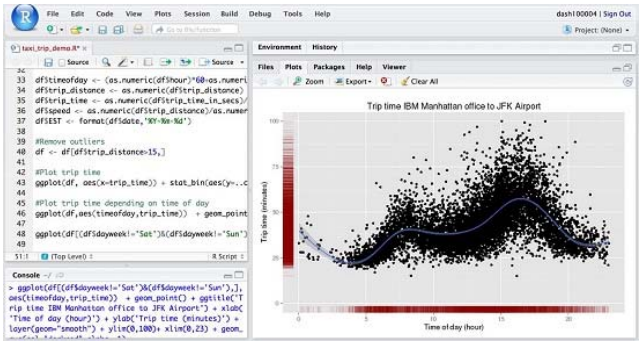

Figure 3 R Studio use with dashDB

5) Geospatial types and functions based on SQL/MM standard

dashDB provides complete coverage of location data types such as points, line strings and polygons along with the full set of geospatial computation and analytic functions as defined by the SQL/MM standard. This is also the basis for seamless integration with state of the art geospatial tooling. Figure 4 illustrates dashDB usage with spatial analytics such as Esri. You can also use the geospatial capabilities either through your own SQL statements or through the above-mentioned R and Python language APIs.

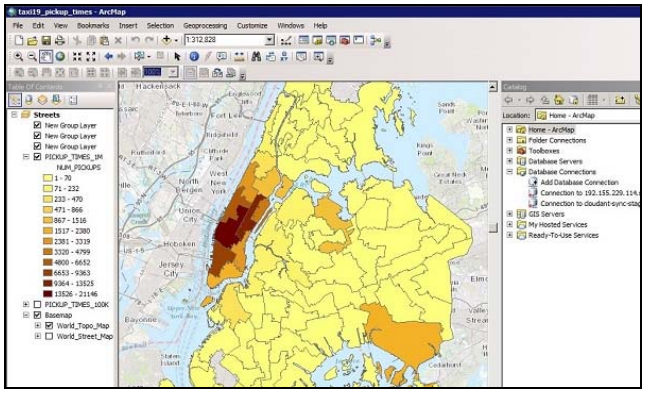

Figure 4 dashDB use with Esri ArcMap

6) Remote Table Access to other data stores

Integrated Fluid Query technology provides key capabilities to unify, fully integrate, and leverage disparate data across Big Data ecosystems. Multiple built in connectors allow you to quickly create a table nick-name to access and query remote database objects from Hadoop data repositories such as Cloudera Impala or structured database objects such as SQL Server, DB2, Netezza, or Oracle, per Figure 5. Common use cases involve queryable archives, bridges to RDBMS islands, Discovery and Exploration, Data Warehouse Capacity relief, transparent data access across your enterprise regardless of location, and unification of Hadoop and structured data stores. This practical use of different data stores can be accessed with existing SQL skills from dashDB.

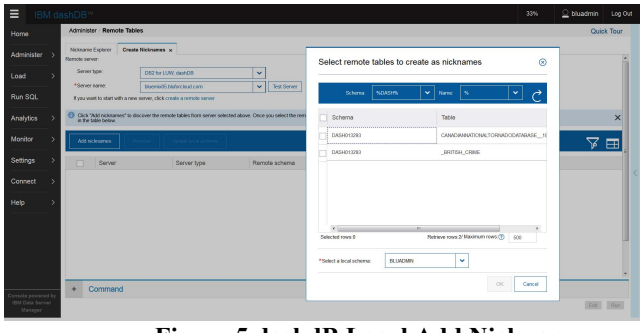

Figure 5 dashdB Local Add Nickname

D. Native Apache Spark Integration

The requirements of Big Data are changing and users are thinking about different ways to process data and therefore looking into open source products like Hadoop or Spark as well as traditional Data Warehouse technology for high performance analytics.

Hadoop and Spark are seen as ecosystems to run various data intensive tasks, especially analytics and machine learning. Some major data warehouse vendors reacted to this trend by extending their offering to include predictive analytics, support for various languages, as R and Python, spatial operations, etc. often summarized as in-database analytics. Nevertheless, most of these implementations are proprietary designs without common usage. The concept of in-database analytics, which brings the compute power to the place where the data is stored, is still important, however the proprietary non-standard APIs to these analytics are limiting.

With dashDB Local we have combined Apache Spark with Data Warehouse in a single engine to provide several key functions. This unique combination relieves many of the pain points caused by an environment where two separate systems process the same data.

1) Simplicity

The first key advantage of the tight combination with Apache Spark and the database is the fact that the system is operational out of the box. This system includes the dashDB database, Apache Spark and a Web console to control and administer the system. Using a Docker image, a user could deploy the dashDB Local container on own hardware without the need of installing and configure many components.

This integration enables a user to begin working on the same data either with the known relational database engine interfaces (e.g. PL/SQL, JDBC) or the new open source Apache Spark analytics ecosystem for more advance and flexible data processing and machine learning.

In this unique combination the user gets a scalable analytic engine that shares the available memory with the database.

dashDB Local provides different methods to leverage the integrated Apache Spark engine:

- REST API interface to run, cancel, or monitor Spark applications in dashDB
- SQL Stored Procedure interfaces to submit or cancel Spark applications
- Interactive Jupyter Notebook to design Spark applications in Scala or Python (separate Docker container)
- Client interface spark_submit to simplify the usage of the REST API interface
- Further prepackaged Stored Procedures which allows to run ready to use analytic algorithms like GLM from within SQL
- One-click deployment to automatically generate and deploy a Spark application from Jupyter notebooks

This means that the integration of Apache Spark can happen on different levels depending on the user's scenarios to start batch, interactive or streaming jobs.

When setting up an Apache Spark Cluster by your own there is no user security included. In contrast IBM dashDB Local ensures that for each user Apache Spark starts an own Spark Cluster Manager so that different users could not see what other users are doing.

As the data is typically stored within the database the Spark jobs of different users could only get the data according to the database privileges. No special security considerations need to be taken.

2) Performance and scalability

The second key area to consider is performance and scalability. A design paradigm for the solution is that for each database node an own Apache Spark cluster is available which fetches the database data collocated using an optimized data transfer. Due to the very tight coupling of analytic computation with the MPP scale-out architecture and the data locality of Spark to the database nodes the same scalability curves normally achieved only in a highly optimized data warehouse for Big Data SQL workloads can now be achieved on Apache Spark.

Figure 6 shows the architecture of dashDB Local combined with Apache Spark. The main controller for each request to Spark is the Spark Dispatcher. The Dispatcher takes care that for each user a different Spark Cluster Manger gets created and that Spark only gets the memory configured.

Each Spark Worker fetches the data collocated to a local shard. Different driver modes allow the user to modify performance characteristics.

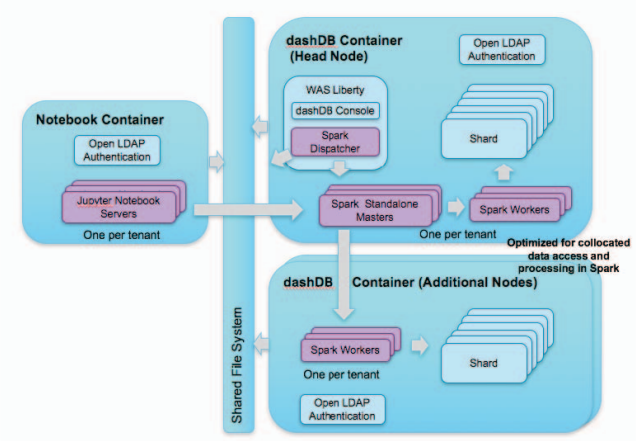

Figure 6 dashDB and Apache Spark

Per default a socket communication is used between the database process and the Spark process. This reduces the memory overhead that would be incurred using a shared memory communication strategy.

As illustrated in Figure 7, the data transfer from the database to Spark gets triggered on the Spark side using JDBC. To optimize the transfer an additional where clause could be pushed to the database to transfer only the data really needed.

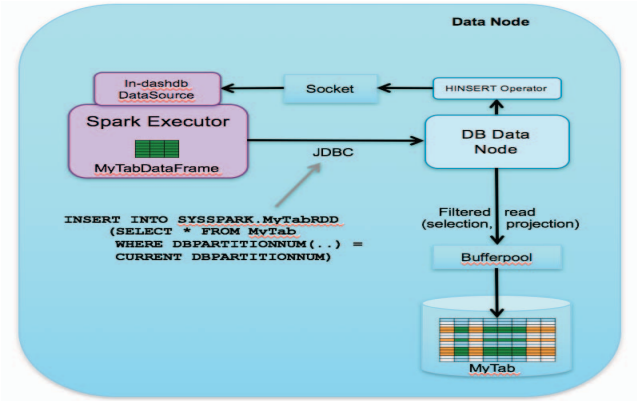

Figure 7 Communication details

3) Open Platform and Ecosystem

Spark is an evolving open platform with a rich ecosystem. It already includes components like Streaming, Machine Learning and GraphX. It also provides several language APIs to use Spark like R, Python, Scala, Java or SQL. The number of features in the Spark ecosystem is rapidly growing.

Providing an out of the box solution with regular updates allowing user to switch to the newest Apache Spark version enables users to create custom analytic solutions based on an industry standard framework instead of a proprietary framework.

A ready to use Apache Spark environment along with Jupyter Notebooks is available in the cloud on datascience.ibm.com, as shown in Figure 8. This allows you to develop your analytics interactively in the cloud using data stored there. You can then export the notebooks and import them locally in the Jupyter notebook environment provides for dashDB Local and run them on premise and operationalize them into deployed applications using the one click deployment option.

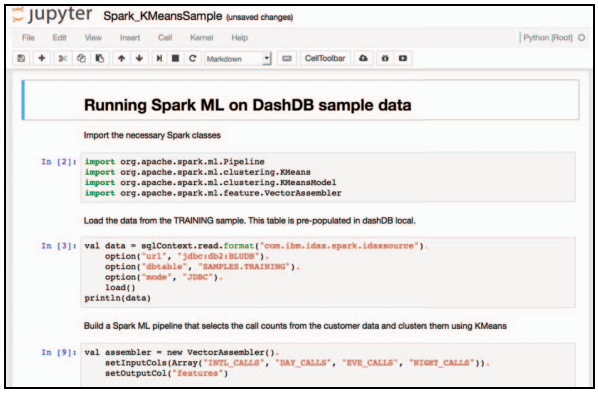

Figure 8 Jupyter notebook

As Apache Spark provides a wide range of data drivers to access different kind of data. Leveraging this capability allows the user to get data into the Data Warehouse. An example of this is the object store driver for Spark. It can be used to integrate data from SoftLayer, Swift, IBM Bluemix Object Service or Amazon S3. This driver can be used to either to store it within the database or just to fetch it for analytic processing jobs. Similarly data in motion using Spark streaming can be used to get data into your Data Warehouse.

E. Native Elasticity and Availability

A dashDB Local cluster is designed to run on a distributed file system provided by the user. Data is sharded (hash partitioned) into the storage onto a number of shards that is several factors larger than the number of servers, though not larger than the number cumulative of cores in the cluster. The association of shards to cores is designed to create a shared nothing topology where memory for each shard is fixed, and associated with a specific subset of the cores in each server. However, this association is fixed only during steady state operations, and can be easily adjusted. The system is design to scale efficiently using cores and memory whether the number of cores per shard is small (e.g. 1 or 2) or large (e.g. 64 or 128). Thanks to this dynamism, the number of cores associated with each shard can be adjusted along with a concomitant modification in the query parallel per shard. Although all files associated with the shard resides on a shared file system, each shard has its own file set that is not shared. Because the system is based on a clustered file system, it is similarly possibly to re-associate shards from one host to another. Each shard is also not tied to or bound to a container that resides on a host. These flexibilities in the architecture enable HA and elasticity. dashDB provides automatic service level HA inside each container. Any service outage or disruption will restore the service in its original container if the container and the host it resides on are still available.

If a server host fails or if the container on that service is no longer available, all services and the shards associated

with that container or host are reassociated with the surviving containers running on other server hosts. The query parallelism per shard is reduced accordingly, as is the memory allocation per shard. Consider the example in Figure 9, for a cluster of four servers. Each server in this example has 6 hash shards of data. In the event of an outage on server D, the shards associated with that server are easily reassociated with the surviving nodes, A, B, C that now services 8 shards each. The cluster continues as a well-balanced unit, albeit with fewer total cores and less total RAM per byte of user data.

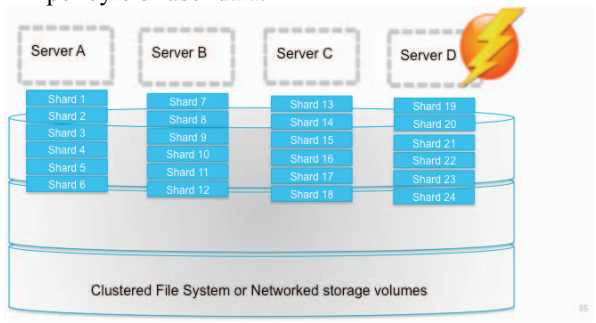

Figure 9 HA for dashDB Local

To achieve elastic contraction the same process is used, except with a deliberate action, rather than an unexpected outage. A server is removed from the cluster, shards are reassociated to the remaining nodes, with adjustments to RAM distribution and query parallelism per shard.

Finally, the process of elastic growth is also very similar to the path of reinstating a repaired node. A new node is added, shards are reassociated, RAM and query degree are both increased per shard. All of these operations are largely automated, though the user does need to provide the new hardware and indicate the requested expansion. And finally full portability of the dashDB stack and all of its data is now simplified using the same mechanics of failover/scale in and restore/scale out. All persistent data including userid, database monitoring history, and your database data is stored in the clustered filesystem.

By copying/moving the clustered file system by any method available to your infrastructure you can now docker run and deploy quick and easily against an entirely new set of hardware with a different physical cluster topology of your choice. dashDB Local is designed for Cloud snapshot/availability zones capabilities to make portability/DR simple and easy.

F. Cloud compatibility with dashDB as a service

dashDB is a family of data repositories. On the public cloud IBM provides dashDB cloud data warehouse (currently on both IBM Bluemix and Amazon AWS), which has a common query engine and user console with dashDB Local. While dashDB Local is designed for fast and simplified deployment with automatic adaptation to user provided hardware, dashDB cloud data warehouse on the public cloud is designed as a fully managed service, where IBM provides configuration, tuning, administration, through automation or DevOps staffing. The common engine components for query processing, language support, console, workload managements, enable near perfect portability analytics code across these domains.

III. DASHDB LOCAL EXPERIMENTAL MEASUREMENTS

We have performed 4 evaluations of dashDB runtime performance. First a serial test of a customer workload over 25TB of data including several thousand customer provided queries used for a large-scale financial analytics. The database had 9 schemas with 1,640 tables and 71,145columns. The data compress to about 9TB on both the appliance and the dashDB Local system. The workload selected comprised of over 250K queries. The queries were:

- 86537 INSERT
- 55873 UPDATE
- 46383 DROP
- 44914 SELECT
- 25572 CREATE
- 2453 DELETE
- 12 WITH
- 12 EXPLAIN
- 5 TRUNCATE

The single stream query performance was compared on each system. Of the entire workload a subset of 15,000 queries were used. Measurements were taken from the 3,500 longest running queries. We compared this to a high performance analytic appliance with similar compute capacity. The dashDB Local system realized an average increase of 27.1 times faster with a median performance improvement of 6.3 times.

Second, the actual concurrent workload was executed as it would execute on a live system. Test described above was run in a multi-user concurrent environment that included executing the workload exactly how they are executed in customer environments. The workload included up to 100 concurrent streams related to various query operations. This resulted in dashDB executing the whole workload in less than half the time, a 2.1x performance improvement with dashDB Local compared to the baseline run on the hardware appliance.

Third, we tested dashDB Local using TPCDS queries [24], and compared these to a high performance analytics appliance.

Fourth, we ran a throughput test of dashDB running on the Amazon Cloud AWS, executing a 5-stream workload of IBM BD Insight workload and compared these results to a

popular cloud data warehouse [1] running on the same platform with identical hardware.

Table 1 summarizes the results. In Test 1, dashDB Local achieved an average query speedup of more than 27x, (and a 6.3x median query speedup) using only slightly more cores. In Test 2, the concurrent workload, realized a 2.1x execution time improvement. In Test 3 running TPCDS queries, dashDB achieved a better than 2x average query speedup. Finally in test 4 comparing against another popular MPP shared-nothing column store with a memory cache on virtually identical hardware dashDB achieved a 3.2x throughput advantage.

Table 1 Workload performance tests on dashDB

	Test 1	Test 2	Test 3	Test 4
Description	Customer workload dashDB vs Appliance (query performance)	Customer workload dashDB vs Appliance (throughput, queries & load)	TPCDS Benchmark dashDB vs appliance	BD Insight Benchmark on Amazon AWS dashDB vs competitor
dashDB hardware	4-nodes X 20 cores 4X 256GB RAM 28TB SSD	4-nodes X 20 cores 4X 256GB RAM 28TB SSD	6 nodes x 24 cores 6X 512 MB RAM 34TB SSDs	32 vcpu 244 GB RAM EBS volumes with 1800 IOPs
Baseline hardware	4-nodes X 16 cores 8x FPGAs 4x132GB RAM 23TB HDD	4-nodes X 16 cores 8x FPGAs 4x132GB RAM 23TB HDD	7-nodes x 20 cores 14x FPGA 7x 132GB RAM 46TB HDD	32 vcpu 244 GB RAM 2.56TB SSD
Performance	**Avg Query Speedup: 27.1X**	**Workload time: 2.1X**	**Avg Query Speedup: 2.1X**	**Throughput increase Qph: 3.2X**

IV. Conclusion

In this paper we have introduced dashDB Local, a new addition to the dashDB family of products. This new engine for Big Data introduces 5 new benefits of 1. Fast fully configured deployments of dashDB Local to Big Data clusters from gigabytes to petabytes making it possible to deploy fully configured clusters for advanced data warehousing and Spark in minutes on clusters of a wide range of sizes and components. 2. Superior workload performance, often several times faster that leading Big Data and Data Warehouse technology, including other columnar technologies. 3. Rich polyglot language support, supporting many dialects of Big Data, SQL languages such as SQL dialects for Oracle, DB2, Netezza, and PostgreSQL, PL/SQL, SQL PL, Python, PySpark, and R. 4. Full stack integration of Apache Spark into the core processing engine, co-existing with the rich SQL engine over the same data sets. 5. Compatibility with dashDB service in the public cloud for hybrid development, QA and production flexibility.

V. dashDB Local Available for Academic Use

IBM has had a long-standing initiative to bring IBM Software to universities for research and teaching [25]. dashDB Local is available for download from the academic initiative web site, along with many other pieces of software. Furthermore, the EULA in dashDB Local has no DeWitt clause [26], so we welcome experimentation with dashDB Local by students and researchers. Similarly a free version of dashDB is available online [1].

[1] The terms of the license agreement for the non-IBM product under test prevent us from mentioning the specific vendor and product.

VI. Future Work

Future areas of exploration for dashDB Local include:

- Performance enhancements. We believe a further 2x performance improvement remains possible in this technology through algorithmic enhancements.
- Improve support for Schema on Read.
- Support for common Big Data storage formats, such as Parquet.
- Support for Big Data Analytics on JSON data.

Acknowledgment

We with to thank several key contributors of the dashDB Local initiative, including: Prem Yerabothu, Scott Andrus, Maria Attarian, David Kalmuk, Girish Venkatachaliah, Thomas Chu, Matthias Funke, and Mitesh Shah.

References

[1] IBM dashDB http://dashdb.com

[2] Docker https://www.docker.com/

[3] M. Stonebraker et al. C-Store: “A column oriented DBMS”, Proc. of the Very Large Data Bases Conf., 2005.

[4] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, J Dees, “The SAP HANA Database -- An Architecture Overview”, IEEE Data Eng. Bull. 35(1), pp. 28-33, 2012.

[5] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, M. Zwilling, “Hekaton: SQL Server’s memory- optimized OLTP engine”, Proc. of the ACM SIGMOD Conf., 2013.

[6] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mühe, T. Mühlbauer, W. Rödiger, “Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer”, IEEE Data Eng. Bull. 36(2), pp. 41-47, 2013.

[7] D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, D. Wood, "Implementation techniques for main memory database systems". Proc. ACM SIGMOD Conf 14 (4): pp. 1–8 (June 1984).

[8] S. Manegold, P. Boncz, and M. Kersten. “Optimizing database architecture for the new bottleneck: memory access”, VLDB Journal, 9(3), 2000.

[9] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, Narang, and R. Sidle, “Constant-time query processing,” Proc. IEEE Intl. Conf. on Data Engineering, 2008.

[10] Jae-Gil Lee et al. “Joins on Encoded and Partitioned Data”, Proc. of the Very Large Data Bases Conf., 2014.

[11] V. Raman et al., “DB2 with BLU Acceleration: So much more than just a column store”, Procs. of the Very Large Data Bases Conf. 6, 2013.

[12] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, D. Sharpe, “Memory efficient hash joins”, in press, Procs. of the Very Large Data Bases Conf., 2015.

[13] A. Storm, S. Lightstone, “Randomized page weights for optimizing buffer pool page reuse”, US Patent 9,037,803 May 19, 2015 Assignee: International Business Machines

[14] A. Ailamaki, D. J. DeWitt, M. D. Hill, D. A. Wood,“DBMSs on a Modern Processor: Where Does Time Go?” VLDB 1999: 266-277

[15] R. Barber, G. M. Lohman, V. Raman, R. Sidle, S. Lightstone, B. Schiefer, “In-memory BLU acceleration in IBM's DB2 and dashDB: Optimized for modern workloads and hardware architectures”, ICDE 2015: 1246-1252

[16] E. Kwan, S. Lightstone, K. B. Schiefer, A. Storm, and L. Wu, “Automatic database configuration for DB2 Universal Database: compressing years of performance expertise into seconds of execution, Proc. of BTW Conf., pp. 620–629, 2003.

[17] A. Aboulnaga, P.J. Haas, S. Lightstone, G.M. Lohman, V. Markl, I. Popivanov, V. Raman, “Automated Statistics Collection in DB2 UDB”, Proc. Of the Very Large Data Bases Conf. 2004, pp. 1146-1157

[18] Michael Armbrust, Doug Bateman, Reynold Xin, Matei Zaharia: “Introduction to Spark 2.0 for Database Researchers” SIGMOD Conference 2016: 2193-2194

[19] Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki, Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael J. Franklin, Ion Stoica, Matei Zaharia: “SparkR: Scaling R Programs with Spark” SIGMOD Conference 2016: 1099-1104

[20] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, Ameet Talwalkar: “MLlib: Machine Learning in Apache Spark” CoRR abs/1505.06807 (2015)

[21] IBM BLU Acceleration http://ibmbluhub.com

[22] IBM Pure Data for Analytics (Netezza) http://www-01.ibm.com/software/data/puredata/analytics/

[23] IBM BlueMix http://bluemix.com

[24] TPCDS Benchmark http://www.tpc.org/tpcds/

[25] http://www.ibm.com/ibm/university/academic/pub/page/academic_initia tive

[26] http://en.wikipedia.org/wiki/David_DeWitt