Palette: Enabling Scalable Analytics for Big-Memory, Multicore Machines

Fei Chen, Tere Gonzalez, Jun Li, Manish Marwah, Jim Pruyne,
Krishnamurthy Viswanathan, Mijung Kim
HP Labs

ABSTRACT

Hadoop and its variants have been widely used for processing large scale analytics tasks in a cluster environment. However, use of a commodity cluster for analytics tasks needs to be reconsidered based on two key observations: (1) in recent years, large memory, multicore machines have become more affordable; and (2) recent studies show that most analytics tasks in practice are smaller than 100 GB. Thus, replacing a commodity cluster with a large memory, multicore machine can enable in-memory analytics at an affordable cost. However programming on a big-memory, multicore machine is a challenge. Multi-threaded programming is notoriously difficult. Further, the memory design of most large memory servers follows non-uniform memory access (NUMA) architecture. While NUMA-aware programming often leads to high efficiency in analytics tasks, it is usually done in an ad hoc manner.

In this demo, we present *Palette*, an analytics framework that exploits large memory to trade space for time while also addressing the challenges of multi-threaded, NUMA-aware programming. Palette manages multiple, index-like data representations for input datasets. An operator may have multiple implementations, each of which uses a different data representation. Palette uses a cost-based approach to automatically select the fastest one on a given dataset. Palette addresses challenges of multi-threaded and NUMA-aware programming by adapting Hadoop for a single multicore machine and modifying it by considering the characteristics of modern NUMA hardware. Users can write programs using exactly the same APIs as those used in traditional Hadoop, while transparently benefiting from multi-threaded and NUMA-aware infrastructure.

We have developed a research prototype of Palette. Specifically, at SIGMOD we will demonstrate how to (1) create an operator, such as time series similarity search, on Palette, (2) execute the operator with Palette's automatic implementation selection feature, and (3) monitor and compare different operator implementations.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2594509.

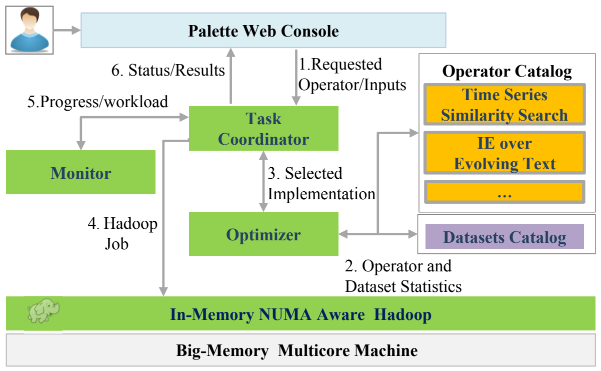
1. INTRODUCTION

Distributed computation platforms such as MapReduce-based Hadoop have become widely popular. These platforms execute massive data parallel computations on clusters of commodity machines. Although these distributed platforms are very appealing to petabyte-scale applications, they need to be reevaluated in light of recent advances in hardware and improved understanding of analytics applications.

Hardware has evolved since MapReduce was introduced in 2004 [5]. In particular, cost of memory and CPU has reduced significantly. In 2004, a single MapReduce node typically had two single-core processors and 2-4 GB of memory [5]. Today, a server with 16 cores and 256 GB RAM costs less than $10,000 USD[1]. Thus, a single server, with CPU resources equivalent to that of a small cluster in 2004, can be equipped, at a reasonable cost, with a much larger memory than the aggregate memory of that cluster. With consolidation to a single machine, network communication is eliminated while the disk I/O costs are substantially reduced, because most of the data can be kept in memory. In fact, in most cases it might be possible to keep *all* the data in memory, given that multiple recent studies have reported small analytics job sizes in practice. For example, 90% of analytics tasks in a production system have input sizes under 100 GB [11]. Similar studies have reported a median task size of only 14 GB [2].

Building general analytics platforms on shared-memory multicore machines presents two challenges. The first challenge is how to provide easy programming on such machines, given that multi-threaded programming is notoriously difficult. Existing frameworks such as OpenMP[2] provide low level APIs, which are hard to use. Furthermore, the trend in hardware has been towards non-uniform memory access (NUMA), where each process can access both its own memory and memory associated with the other processors (remote memory) coherently. However, it is faster to access local memory. While NUMA-aware programming is important in achieving high efficiency in analytics tasks [3], it is done in an ad hoc manner today, where users have to figure out how to localize memory access of each thread and reduce remote memory access as much as possible for each analytics task. The second challenge is to adequately exploit large memory for efficient analytics processing. The abundant memory sometimes can hold more than raw input data. How can we fully leverage additional available memory to speed up processing?

[1]http://www8.hp.com/us/en/products/proliant-servers/product-detail.html?oid=5211699
[2]http://www.openmp.org/



Palette Web Console
6. Status/Results
1.Requested Operator/Inputs
Operator Catalog
Time Series Similarity Search
IE over Evolving Text
...
5.Progress/workload
Task Coordinator
Monitor
3. Selected Implementation
4. Hadoop Job
Optimizer
Datasets Catalog
2. Operator and Dataset Statistics
In-Memory NUMA Aware Hadoop
Big-Memory Multicore Machine

Figure 1: Palette architecture.

To address the two challenges, we have developed *Palette*, an analytics framework providing ease of programming and efficient execution on big-memory, multicore machines. In order for users to easily develop their analytics operators on Palette, we modify Hadoop so that it runs in-memory on a single shared-memory multicore machine and is NUMA-aware. The modified version of Hadoop is compatible with the original Hadoop in its APIs. This allows users to run their existing Hadoop JARs on Palette without any changes. Thus, users can transparently take advantage of multicore and NUMA memory without worrying about painful multi-thread and NUMA-aware programming.

Palette exploits large memory by *trading space (memory) for time*. Each analytics operator, implemented in Palette, materializes and manages multiple, large, index-like auxiliary data representations in order to speed-up its execution. An operator may have multiple implementations, each with different data representations, providing different space-speed trade-offs. Palette uses a cost-based approach to automatically select the fastest implementation of an operator on a given input data instance.

Palette users can easily construct analytics operators that exploit multicore NUMA architecture. They can also easily port existing Hadoop code to Palette with minimal effort. Further, users can construct complex analytics applications by composing existing Palette operators and can share them with other Palette users.

2. PALETTE ARCHITECTURE

The architecture of the Palette system is described in Figure 1. We describe the system components below.

Operator and Dataset Catalog: A single, logical analytics operator in Palette may have multiple implementations, each with a different performance curve and utilizing different representations. We refer to such a logical operator as an "operator family." This is analogous to a DBMS join operator (equivalent to a Palette operator family), which may be implemented as an index-based join, a hash join or a sort-merge join (equivalent to multiple implementations). Palette's operator catalog maintains the list of operator families, their implementations, and the metadata for the operator families and implementations (e.g., cost models). The dataset catalog consists of data definitions and materialized data representations. For illustration purposes, we demonstrate two operator families, but users can easily add others as we will show in Section 3.1.

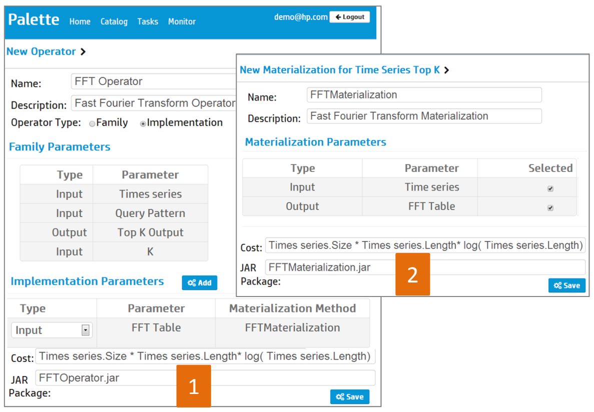
A. Time Series Top-k Similarity Search: This operator searches a time series database to retrieve the k-most similar segments to a query segment. The candidate segments are the sub-segments of the time series in the database. We implement four variants of this operator with different performance characteristics. (1) *Naïve*: This implementation compares the query segment against every time series sub-segment in the database, while abandoning computations on segments when it becomes apparent that they are not among the k most similar segments. The running time of this implementation depends on both the time series dataset size as well as the query segment length. (2) *FFT-based*: The second implementation utilizes a Fast Fourier Transform (FFT) on the time series dataset to rapidly compare every sub-segment of each time series in the database with the query segment. This requires pre-computation and materialization of the FFT values, but the execution time is independent of the length of the input query segment. (3) *Locality sensitive hashing (LSH)-based*, and (4) *Angular hashing-based*: These implementations build a similarity-search index on the candidate sub-segments using LSH [1] and angular hashing [8] techniques respectively. These indices occupy much larger space than both the original database and FFT, but they enable fast retrieval of a small subset of candidate segments with probabilistic guarantees.

B. Information Extraction (IE) over Evolving Text: This operator is based on our previous work [4]. Given an evolving text corpus (such as versioned documents in a content management system), this operator performs IE (e.g. extracting named entities) using Conditional Random Fields (CRFs). We have two implementations. (1) *Naïve*: This implementation reruns the CRF on each snapshot of the evolving corpus. (2) *Incremental*: This implementation applies the CRF over changed text segments between consecutive snapshots. The incremental implementation is much faster than the naïve one when the corpus is only modified slightly between snapshots. It also requires storing by-products of runs and thus significantly larger memory.

Optimizer: When users invoke an operator family on an input dataset, the optimizer performs a cost-based optimization to select the fastest implementation on the given dataset. The cost models are provided by users when they add new operators or implementations. The optimizer inspects the catalogs, enumerates all possible implementations, and estimates their costs (runtime). The estimation is done by instantiating the cost models associated with the implementations based on input data characteristics. Different implementations may require different data representations, and therefore the optimizer must either ensure that the necessary representations are materialized, or determine what materialization steps need to be scheduled prior to invoking the implementations. In the latter case, the optimizer also adds the cost of materialization to the total cost. By automatically performing this cost-based optimization, Palette hides the space-speed trade-offs complexity from the users.

Task Coordinator: This component orchestrates workflows in Palette. For example, once the optimizer selects the fastest implementation, the task coordinator translates the steps required to execute the selected implementation and to materialize any missing data representations into Hadoop jobs, and submits them to the execution engine. Upon completion of the jobs, it presents the results to the users.

In-memory NUMA-aware Hadoop: The unique execution engine of Palette is an *in-memory* Hadoop engine. We modify the Apache Hadoop code base (our current implementation is based on version 1.1.2) to extend its pseudo distributed mode that is originally designed to support only one HDFS data node and one MapReduce task tracker node,



New Operator

Name: FFT Operator

Description: Fast Fourier Transform Operator

Operator Type: Family [✓ Implementation]

Family Parameters

Type	Parameter
Input	Times series
Input	Query Pattern
Output	Top K Output
Input	K

Implementation Parameters

Type	Parameter	Materialization Method
Input	FFT Table	FFTMaterialization

Cost: Times series.Size * Times series.Length* log(Times series.Length)

JAR Package: FFTOperator.jar

New Materialization for Time Series Top K

Name: FFTMaterialization

Description: Fast Fourier Transform Materialization

Materialization Parameters

Type	Parameter	Selected
Input	Time series	✓
Output	FFT Table	✓

Cost: Times series.Size * Times series.Length* log(Times series.Length)

JAR Package: FFTMaterialization.jar

Figure 2: Creating a Palette operator.

so that the entire machine can now support a configurable number of HDFS data nodes and MapReduce task tracker nodes. Each HDFS data node or MapReduce task tracker node is bound to a collection of CPU cores. A Map or Reduce task instance launched from a task tracker node will be bound to the same collection of CPU cores. Such a CPU binding policy is realized through NUMA utilities [10] supported by the underlying operating system. To exploit the available large memory, the local data store of each HDFS data node is bound to a RAM-based file system such as tmpfs, RamDisk or RamFs.

Note that the cluster version of Hadoop looks closely at where the HDFS data blocks are located, and tries to schedule the computation as close to the data as possible. By binding an HDFS data node to a collection of cores using NUMA utilities, in-memory Hadoop is able to preserve Hadoop's data-locality-aware policy, such that Map or Reduce task execution on CPU cores will access data stored at local memory as much as possible. To execute a job, no changes are required to the users' Hadoop code, the Hadoop client utilities or its web-based monitoring tool.

Monitor: Palette includes a monitoring mechanism that tracks the progress, the execution steps, and the resource workload and reports the status to the users. The system also records real-time statistics such as CPU utilization and memory consumption to understand processing patterns.

Putting It All Together: We consider creating and executing an analytics operator on Palette. When users create a new operator family via the Web console (see Section 3.1), the task coordinator adds the operator family, its implementations and their meta data to the operator and dataset catalogs. When users invoke an existing operator family on input data via the Web console (see Section 3.2), the task coordinator passes the invoked operator family and its parameters (e.g., the input dataset) to the optimizer. The optimizer then consults the operator and dataset catalogs to select the fastest implementation, as discussed above. The task coordinator translates the execution steps of the selected implementation and any necessary materialization into Hadoop jobs and submits it to the in-memory Hadoop engine. While the jobs are executed, the job monitor keeps track of their progress and CPU core utilization. The runtime monitoring information is saved and can be used later for comparing the performance of different implementations. Finally, once the jobs are completed, the task coordinator returns the results and job status to users.

3. DEMONSTRATION SETTING

In this section, we describe the three aspects of Palette we propose to demonstrate using time series search operator.

3.1 Creating an Operator Family

We will demonstrate (1) the ease of developing an operator which takes advantages of many cores and NUMA architecture on Palette, and (2) the procedure to create multiple operator implementations and materializations on Palette.

Ease of Programming: To create an operator family, users need to implement the Hadoop JARs for all the operator implementations and materialization methods. We will show the Hadoop source code of one of the time series implementations to illustrate how to program using the traditional Hadoop APIs.

Creating Operator Implementations and Materializations: Once we implement the JAR packages, we need to declare the implementations, their metadata and dependencies. We will demonstrate the step-by-step procedure for declaring them. Palette provides a sequence of wizards for such declarations. The first wizard allows users to declare the operator family, including its name and parameters. For the time series operator, the parameters include the input database, the query, the number of results to be returned and output file. The next wizard allows users to declare the different implementations and materializations.

Figure 2 illustrates the UI for declaring the FFT implementation and materialization. In the FFT implementation declaration wizard, we need to define its name, its dependency on the time series operator family, its parameters, cost model and JAR package (which has been developed in the previous step). Most of its parameters are inherited from the operator family. It also takes one additional parameter, which is the FFT of the time series in the original database. The next wizard (see the sub-figure 2 in Figure 2) allows users to declare the FFT materialization method. It is very similar to the implementation declaration wizard. Notice that it has its own cost model and JAR package. The Palette optimizer is able to take into account the cost of materializing FFT along with the cost of invoking the FFT implementation if the FFT has not been materialized.

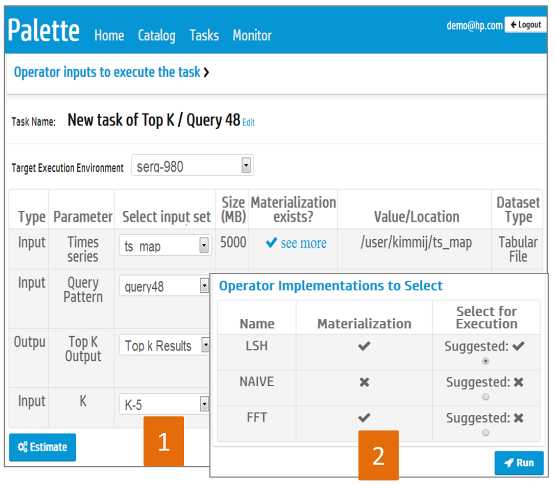
Palette captures the dependency and metadata associated with the operator family, implementations and materializations, and stores them in the catalogs. It then uses the captured dependency and metadata to automatically manage materializations (e.g. checking if FFT on a given time series database instance has been materialized) and automatically pick the fastest implementation.

3.2 Executing an Operator Family

We will demonstrate (1) how to execute a Palette operator family, (2) Palette's ability to manage materializations and automatically select the fastest implementations, and (3) Palette's in-memory Hadoop compatibility with JAR codes written for the cluster version of Hadoop.

Executing an Operator Family: Palette provides a step-wise wizard (see Figure 3.(a)) to execute an operator. First, the user selects the operator family to execute, and fill in the parameters (e.g., the input). Furthermore, the "Target Execution Environment" choice in the wizard allows users to choose between a cluster and a multicore computing environment to execute the operator.

Managing Materializations and Automatic Optimization: Once the user clicks on the "Estimate" button, the optimizer evaluates different implementations of the time series operator family and recommends the fastest imple-



Operator inputs to execute the task ›

Task Name: New task of Top K / Query 48 Edit

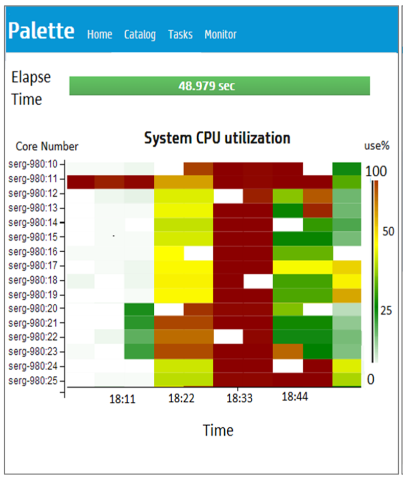
Target Execution Environment: serg-980

Type	Parameter	Select input set	Size (MB)	Materialization exists?	Value/Location	Dataset Type
Input	Times series	ts map	5000	✓ see more	/user/kimmij/ts_map	Tabular File
Input	Query Pattern	query48				
Outpu	Top K Output	Top k Results				
Input	K	K-5				

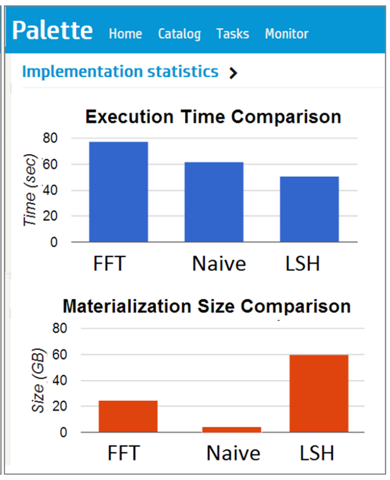
Operator Implementations to Select

Name	Materialization	Select for Execution
LSH	✓	Suggested: ✓ ●
NAIVE	✖	Suggested: ✖ ○
FFT	✓	Suggested: ✖ ○

(a)



(b)



(c)

Figure 3: (a) Executing an operator, (b) Monitoring CPU core utilizations (showing core 10-25), and (c) Comparing performance.

mentation. Along with the recommendation, if an implementation uses auxiliary data representations other than the input dataset, Palette also shows whether they are materialized (see the sub-figure 2 in Figure 3.(a)). Next, the user invokes one of the implementations. When the selected implementation is executed, Palette visualizes the progress of the Hadoop job and the CPU core utilization (see Figure 3.(b)).

To show why we need multiple implementations, we will provide input datasets with different characteristics (e.g., varying length of time series queries) so that a different implementation is faster for each input. This will demonstrate that the optimizer automatically chooses the fastest one.

In-memory Hadoop Compatibility: We will provide two computing environments at SIGMOD. One is a small cluster of eight nodes, each with four cores and 16 GB memory. The other is a multicore big-memory machine with 32 cores and 1 TB memory. The cluster version of Hadoop and in-memory Hadoop are installed on the computing platforms respectively. When invoking an implementation, SIGMOD attendees can choose between these two computing environments. In the backend, the same JAR file will be executed.

3.3 Comparing the Speed and Space

Palette provides tracking features that record history of runtime execution and monitoring. The SIGMOD attendees can select from different implementations which have been executed and compare their execution time and space. Figure 3.(c) shows the query times and memory consumed for three time series search implementations (naive, FFT-based and LSH-based) for a given query and time series database. The LSH implementation is the fastest and consumes the largest space. This exemplifies the space-speed trade-offs that Palette aims to achieve.

4. RELATED WORK

Recently, there has been interest in in-memory computing in DBMS [7, 6] and analytics platforms [14]. Some of these focus on moving MapReduce based platforms to memory. However, most of them [14, 12] still target cluster based distributed memory computing environments. In contrast, Palette targets single machine shared-memory computing environments, which is complementary to these works. Existing research on shared-memory MapReduce platforms [13, 9, 2] do not consider either NUMA architecture or fully leveraging available memory to speed up execution.

5. CONCLUSION

We have presented Palette, a framework for developing scalable analytics operators for big-memory multicore machines. It enables easy multithreaded, NUMA-aware programming by using in-memory NUMA-aware Hadoop as its execution engine. It enables space-speed tradeoffs by managing auxiliary data representations for multiple operator implementations and using a cost-based optimizer to choose the fastest one, thus fully exploiting large memory.

6. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *CACM*, 51(1), 2008.

[2] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for Hadoop: Time to rethink. *SOCC-13*.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB-13*.

[4] F. Chen, X. Feng, C. Ré, and M. Wang. Optimizing statistical information extraction programs over evolving text. *ICDE-12*.

[5] J. Dean and S. Ghemawat. MapReduce: Simplied data processing on large clusters. *OSDI-04*.

[6] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. *SIGMOD-13*.

[7] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: Data management for modern business applications. *SIGMOD Record*, 40(4), 2012.

[8] Y. Gong, S. Kumar, V. Verma, and S. Lazebnik. Angular quantization-based binary codes for fast similarity search. *NIPS-12*.

[9] K. A. Kumar, J. Gluck, A. Deshpande, and J. Lin. Hone: Scaling down Hadoop on shared-memory systems. *VLDB-13*.

[10] C. Lameter. An overview of non-uniform memory access. *CACM*, 56(9), 2013.

[11] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using Hadoop on a cluster. *HotCDP-12*.

[12] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta. M3R: Increased performance for in-memory Hadoop jobs. *VLDB-12*.

[13] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. *IISWC-09*.

[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud-10*.