Integrating query processing with parallel languages

Brandon Myers
University of Washington
bdmyers@cs.washington.edu
July 2016
Supervised by Mark Oskin and Bill Howe

***Abstract*—In this thesis we propose new techniques for using parallel languages to improve query processing. Optimizing a query plan *and* its particular implementation is important for efficient processing on modern systems. First, we present our work on a parallel representation of queries using *partitioned global address space* languages that enables new optimizations. Next, we propose future work on cooperative optimization of query plans and imperative programs in the context of parallel applications that include queries.**

I. Introduction

For high performance query processing, multiple layers of the compute stack must be considered. Recent systems have demonstrated orders of magnitude performance improvements over conventional interpreted, iterator-based query processing by specializing the implementation of a query plan before execution [4], [22], [26], [29].

Although distributed memory clusters are important for scaling up performance on large query workloads, the principle of multi-level optimization has not been applied in this context. Query systems that generate *distributed* programs do so by generating program fragments for individual processors and stitching them together with communication calls [12], [28], [24], confounding further optimization of the whole program. We hypothesize that important optimizations can be applied on an intermediate representation (IR) of the query that is *both* parallel, like a query plan, and detailed, like a source or bytecode program.

The completed contributions of this thesis are:

1) We designed new techniques for compiling queries into fast code for distributed memory systems by including a *partitioned global address space* (PGAS) IR; this enables new optimization opportunities. Pipelines of operators are converted into parallel loops that can be optimized as a whole, even across communication boundaries. We implemented the techniques in RADISH, a query compiler that translates relational queries into PGAS code that runs on distributed memory systems. To the best of our knowledge, RADISH is also the first query processing system that integrates with PGAS languages.
2) We built a PGAS language and runtime, Grappa, and generated programs for it using RADISH. GRAPPA's runtime features, like message buffering, task inlining, and lightweight task scheduling mitigate the overheads of fine-grained, data-centric code that RADISH emits. The system executes queries $12.5\times$ faster than Shark, which has been shown to have performance comparable to parallel analytical databases. Although we only evaluate code generated for GRAPPA, RADISH is extensible to target other PGAS languages.

Queries are often components of larger applications. Achieving the highest performance on these applications will require optimizing queries within the context of a parallel program. Existing work on integrating queries in parallel languages has been restricted to *data parallel* programming models like Dryad [35], MapReduce [30], [14], [36], and Spark [34]. Many efficient parallel programs rely on fine-grained manipulation of shared data and *task-level parallelism*; these features are provided by high-performance parallel languages, including PGAS.

We propose the following additional contribution for this thesis: We will explore how query plan optimization and optimization of PGAS programs can be composed. There are two directions that the research can take. 1) Explore the benefits of a query optimizer that can use a PGAS compiler to inform it about UDFs, or conversely, that can pass assertions to the PGAS compiler based on data statistics and algebraic semantics. 2) Explore the benefits of using the PGAS application context of language-integrated queries for informing the query optimizer.

II. Radish: an IR for optimizing distributed query plans

We identified a shortcoming in all existing distributed query processing systems that generate code. These systems compile plans for individual processors and stitch them together with communication calls. This approach misses important optimizations because it throws away information about parallelism. We hypothesize that those optimizations can be applied on an IR of the query that is both parallel, like a query plan, and detailed, like a source or bytecode program.

In [1], we explore this hypothesis by translating queries into a program in a PGAS language. The key feature of a PGAS language is that memory is explicitly partitioned so that memory access to different regions has different costs. This approach unlocks a collection of optimizations that are accessible to neither query plan optimizers nor sequential language compiler.

A. Background: PGAS languages

PGAS languages are the dominant shared memory languages for programming distributed memory clusters (Figure 1). Their critical attribute beyond traditional shared memory programming models is an explicit partitioning of the shared address space across nodes of the cluster. Partitions allow the programmer and language compiler to reason about locality for a non-uniform memory access (NUMA) architecture. PGAS languages enable higher productivity than message passing libraries and in some cases better performance due to a distributed-aware compiler and runtime optimizations enabled by language support for parallelism and data distribution [13], [5], [6], [7], [18]. For the techniques in our work, we assume that a target PGAS language provides the following constructs beyond those of a conventional shared memory language:

1) concurrent tasks: `spawn t` forks a new asynchronous task whose reference is `t`; `s.sync` blocks the calling task until task `s` is finished
2) parallel for loops: `parallel forall i in I: body` runs an iteration of `body` for each element in the collection; iterations only *may* be run in parallel, so they are allowed to synchronize with each other but may not have inter-dependences
3) control over data partitioning
4) task movement: `on partition(L): body` indicates that the calling task must move to the partition `L` before executing `body`; `L` may be a constant expression or dynamically evaluated expression of type `Integer`.

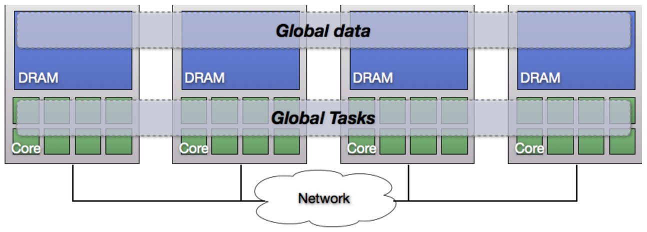


Fig. 1: The global-view PGAS programming model on a distributed memory cluster. Memory is abstracted as a shared, partitioned heap and concurrency is expressed with tasks.

B. Motivating example

We illustrate in an example how using a PGAS representation of the query will enable a unique optimization. Consider the SQL query in Figure 2a (top). We use the query execution plan in Figure 2b where the join of R and S is performed by hash partitioning both tables on key b. The code generated by current systems for probing a hash table of S tuples with tuples of R is in the form of Figure 2a (middle). The multiplication could be performed in producer task before line 4 or in the consumer task as shown on line 8, but the compiler has too little knowledge to perform such reasoning, so this decision can only be made earlier: in the query optimizer. However, at the level of the query plan, there is no obvious difference between these two choices; in either case, two numbers will be sent over the network: `(r.a*r.b, r.b)` in one case, and `(r.a, r.b)` in the other.

Now consider the PGAS implementation in Figure 2a(bottom). From this code, the PGAS compiler is able to explore an additional class of decisions related to distributed execution on the target machine. The on partition (line 2) indicates that the execution of the loop iteration must move to the partition specified by the value in brackets. In this case it is the partition corresponding to the hash of the value r.b. The parallel language compiler that reasons over the PGAS code a detailed, imperative IR of the query will consider the likelihood that the multiply functional unit is available to choose the best place for the multiplication: either at line 2 or on the other partition after line 3. This kind of optimization is inaccessible to both a database-style algebraic optimizer that cannot reason at the level of detail of instructions and an ordinary shared memory compiler (LLVM, JVM, .NET) that cannot reason about partitions.

C. Generating PGAS programs

We extend techniques for pipeline-based code generation for query plans [26] to produce distributed parallel programs. RADISH parallelizes query execution using tasks and shared memory in three ways. First, RADISH reduces communication and supports fine-grained updates via careful layout of global data structures: memory locations that are accessed together are placed in the same partition. Second, RADISH considers diverse space of plans by considering operators that involve fine-grained (tuple granularity) and coarse-grained (relation granularity) synchronization between pipelines. Third, RADISH produces efficient, data-centric parallel code for each pipeline. Within a processor, data-centric code makes efficient use of memory bandwidth by sending one tuple through a pipeline of operators at a time. Since the PGAS compiler understands communication, its optimization window extends across a whole pipeline regardless of communication boundaries.

III. GRAPPA: DISTRIBUTED SHARED MEMORY FOR HIGHLY CONCURRENT APPLICATIONS

Data-intensive applications with random access patterns pose a challenge for distributed memory platforms. To tolerate even the relatively low microsecond latency of RDMA network operations requires significant memory concurrency. Having and managing significant concurrency is not enough. Network interfaces also do not have sufficient injection rate to use the full bisection bandwidth of the network under a load of small messages. These challenges complicate the NUMA shared memory machine model for distributed memory. In addition to synchronization and locality, a programmer must also avoid small messages. Compilers have addressed this gap with a variety of communication avoiding, overlapping, and coalescing optimizations [3], [2], but they are currently limited in the scope of optimizations.

We built Grappa [25], a global-view PGAS runtime for C++11, to address these limitations. It is designed to have an execution model that is efficient for distributed memory systems while not being onerous to programmers. To do this it comprises 1) lightweight tasks to manage enough concurrency to overlap network requests, 2) execution moving to eliminate network hops, and 3) an aggregating network stack to increase small message performance. These runtime features of Grappa are important for RADISH, whose task-per-tuple execution model generates many tasks and small network messages.

IV. OPTIMIZING QUERY PLANS AND PGAS PROGRAMS

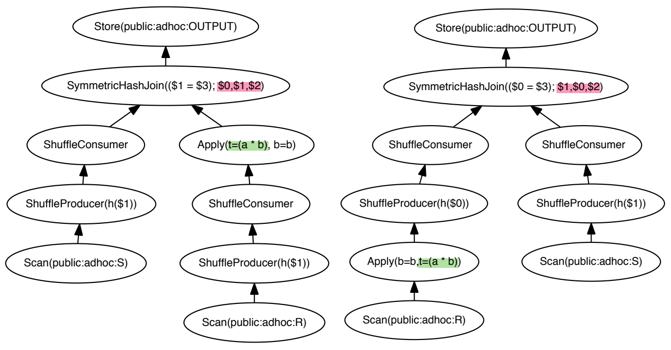
RADISH demonstrated the benefit of introducing a low-level parallel IR into query plan evaluation. It is also the first step towards integrating query processing with PGAS

```
1  -- SQL query
2  SELECT R.a*R.b, R.b, S.b
3    FROM R,S
4    WHERE R.b=S.a;
```

```
1  // Existing solutions
2  submit task:
3    local forall r in R:
4      push r to hash(r.b)
5
6  submit task:
7    while(r = pull()):
8      t = r.a*r.b
9      local forall s in hashtable.lookup(r.b)
10       emit t, r.b, s.b
```

```
1  // Using PGAS as a query IR
2  submit task:
3    global forall r in R:
4       on partition [ hash(r.b) ]
5        t = r.a*r.b
6        global forall s in
             hashtable.lookup(r.b)
7          emit t, r.b, s.b
```

(a)



(b)

Fig. 2: Why using a low-level parallel language as an intermediate query representation is useful. (a) (top) SQL query to evaluate. The middle and bottom listings show concurrent program implementations of the probe side of the join. In code generated by existing systems, the compiler cannot reorder the multiplication (shown at line 8) between disparate parts of the program. In the PGAS code generated by Radish, the compiler can now reorder the multiplication. (b) Two candidate parallel query plans. Left applies the multiplication (in green) after hash partitioning and right applies the multiplication before hash partitioning. A conventional query planner has no basis to pick between these plans: both have the same cost as the number of attributes sent over the network (in pink) is equal.

languages. However, so far the integration has been quite loose: RADISH only adds code generation to link a conventional query optimizer with a PGAS compiler. Many applications are built as complex queries comprising UDFs or general purpose code with language-integrated queries. We will discuss opportunities for improving program optimization in each of these situations.

A. Queries with UDFs

1) Related work

User-defined functions (UDFs) are written in an imperative language rather than the query language. Query optimizers that reason over only a known set of physical operators and treat UDFs as black boxes may pick poor query plans. A number of research efforts have analyzed UDFs to improve query planning. These efforts have addressed two challenges: integrating UDFs into query optimization and analyzing UDFs to determine important properties.

Chaudhuri et al. and Hellerstein present algorithmically efficient optimization techniques for query plans that include UDFs that are expensive to evaluate [8], [17]. Hellerstein's approach requires estimation of computation cost and selectivity but leaves estimation as future work.

Including operators with UDFs in query optimization requires that their properties be known. Manimal [21] and HadoopToSQL [20] use simple static analyses on map UDFs in MapReduce jobs to infer selections and projections to apply database-style optimizations. Specifically, Manimal reduces usage of disk bandwidth using relational indexing and column-oriented storage and HadoopToSQL executes the extracted query on a database. Hueske et al. [19] presents techniques to automatically reorder a plan of "MapReduce-style" UDFs using static analysis and reordering proofs. Tupleware [12] is initial work that uses LLVM to analyze UDFs to estimate two factors for an operator: compute time and load time. The query planner uses a simple comparison of compute time and load time to decide whether a map operator should be pipelined or vectorized. GraphX [15] dynamically uses the JVM bytecode introspection to find unreferenced fields to remove joins.

2) Proposed research

Implementing analyses of UDFs in the style of Tupleware in a Radish query would enable better cost-based optimization of query plans. However, we hypothesize that exhaustively adding the extraction or annotation of more properties of UDFs to the cost estimation is an unscalable effort. A complimentary approach suggests a new research question. In a system like Radish that generates intermediate code, a complimentary approach would be to provide more information to the intermediate compiler to allow it to apply its optimizations more effectively. Specifically, we could empower the intermediate compiler by providing it more complete semantic information about the program generated from the query. Two types of useful information that the query planner can provide are (1) data statistics and constraints and (2) semantic behavior of data structures.

Data statistics and constraints would allow the IR compiler

```
1  forall r0 in R {
2    // r0 is only being stored for use elsewhere
3    hash.insert(r0[2], r0)
4  }
5
6  forall s in S {
7    forall r1 in hash.lookup(s[2]) {
8      // the query planner knows this r1 is r0
9      // and is the only use of r0
10     emit my_udf(r1, s)
11   }
12 }
```

Fig. 3: Generated code for hash table join. The knowledge that tuples `r0` are entering and exiting a join is lost in the program. If the compiler has this knowledge then its optimization window could extend from reading r0 in line 1 to using it in line 10.

to infer constants or optimize based on stronger symbolic constraints for variables. For eample, if the compiler knows the bounds of loops then it can perform aggressive unrolling and blocking optimizations. Some limited types of assumptions that would be useful work with existing compilers [27].

Properties of operators that were provable in the query plan are lost during translation to the IR program. An example is shown in Figure 3. In a hash table join, the build tuples actually are stored into the hash table and are only accessed when they are looked up by the probe code. The link between the insertion and use of build tuples is lost because the compiler cannot prove that the probe pipeline is the only site where lookups occur. Restoring this link in the generated program widens the optimization window for the compiler. Approaches may draw inspiration from work like the Broadway Compiler [16], which accepts libraries annotated by an annotation language that summarizes effects of procedure calls.

Interesting research questions are (1) how do you pass semantic information down in a way that is maintainable and flexible? (2) what is the best assignment of concerns between the query optimizer/code generator and the intermediate compiler? (3) what optimizations are unlocked by larger windows and how much benefit is gained?

B. *Programs with language-integrated queries*

1) *Related work*

Declarative queries and imperative programming languages differ in several ways: how they are optimized, how they are expressed and tuned, and in their approaches to data types and modularity [11]. The term “impedance mismatch” [23] usually refers to the inefficiencies in object-relational mappings but also applies to these other differences. This mismatch remains an open problem but there are numerous relatively successful integrated systems, as surveyed by Cook and Ibrahim [11]. We focus on work that optimizes across programs and integrated queries, rather than techniques that optimize queries in isolation (e.g., [24] and [31] optimize individual LINQ queries).

Weidermann et al. [32], [33] use static analysis of programs with implicit database interaction to extract more efficient queries. Their approach uses an operational semantics of the host language for discovering data traversal paths and conditions in the program. These paths and conditions are combined to form a database query. The technique supports translating joins that are generated by iteration of an object field (e.g., for each employee print their managers name) but does not infer joins from conditions in the code. Cheung et al. [10] achieve a similar goal and infer more queries. The authors take a different approach: their technique generates Hoare style verification conditions for a code fragment and then uses constraint-based synthesis to find a relational algebra expression that satisfies the conditions. This technique is able to translate looping code beyond just iteration: it handles joins derived from program conditions, as well as aggregates. So far no such system supports updates or insertions to the database. There are further research opportunities in enhancing the query optimizer with information from the application context [9].

2) *Proposed research*

We hypothesize that consideration of application context for query optimization becomes even more important in distributed programs. To avoid costly data movement, the query optimizer needs to consider the distribution of the input and output data. This sort of information is available in a conventional parallel database in the form of metadata about partitions and clustered indexes.

Since PGAS languages are partition-aware, this flavor of information is available to feed to the query engine. However, this data distribution information typically stops short of semantics of data structures. For example, an input table to a language-integrated query is might already in the form of an index (e.g., in a distributed hash table). Interesting research questions are (1) what kinds of contextual information from a parallel application are useful for optimization of integrated queries? (2) how can we automatically infer structural information about distributed data structures that aids query optimization? (3) how do we balance the high-performance programmers intuition with portability and the query optimizers decisions? (4) what is the most effective way to profitably pick a distribution for optimizing the whole application?

V. Conclusion

Building efficient data processing systems requires effort at multiple levels of the compute stack. In this thesis we explore how flexible parallel languages and query optimizers can cooperate to generate efficient parallel programs. We draw from work in both data management and programming languages, especially on specializing query plan implementations and static analyses of imperative programs. Our work impacts programming productivity and optimization of high-performance parallel applications that involve *both* irregular and data parallel computation.

References

[1] This paper is currently under blind review.

[2] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. Communication optimizations for distributed-memory X10 programs. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1101–1113, May 2011.

[3] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, April 2006.

[4] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[5] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel Language. *International Journal of High Performance Computing Application*, 21(3):291–312, Aug. 2007.

[6] B. L. Chamberlain, S. eun Choi, S. J. Deitz, and L. Snyder. The high-level parallel language ZPL improves productivity and performance. In *In Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[8] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 87–98, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[9] A. Cheung. Towards generating application-specific data management systems. In *CIDR*, 2015.

[10] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 3–14, New York, NY, USA, 2013. ACM.

[11] W. R. Cook and A. H. Ibrahim. Integrating programming languages & databases: Whats the problem? *Expert Article*, 2005.

[12] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: Redefining modern analytics. *CoRR*, abs/1406.6667, 2014.

[13] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2005.

[14] E. Friedman, P. Pawlowski, and J. Cieslewicz. Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment*, 2(2):1402–1413, 2009.

[15] J. Gonzalez, R. Xin, and A. Dave. GraphX: graph processing in a distributed dataflow framework. *Proceedings of the 11th Operating Systems Design and Implementation*, 2014.

[16] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. *ACM SIGPLAN Notices*, 35(1):39–52, 2000.

[17] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Trans. Database Syst.*, 23(2):113–157, June 1998.

[18] P. N. Hilfinger, D. O. Bonachea, K. Datta, D. Gay, S. L. Graham, B. R. Liblit, G. Pike, J. Z. Su, and K. A. Yelick. Titanium language reference manual, version 2.19. Technical report, UC Berkeley Tech Rep. UCB/EECS-2005-15, 2005.

[19] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *Proc. VLDB Endow.*, 5(11):1256–1267, July 2012.

[20] M.-Y. Iu and W. Zwaenepoel. Hadooptosql: A mapreduce query optimizer. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 251–264, New York, NY, USA, 2010. ACM.

[21] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, Mar. 2011.

[22] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[23] D. Maier. Advances in database programming languages. chapter Representing Database Programs As Objects, pages 377–386. ACM, New York, NY, USA, 1990.

[24] D. G. Murray, M. Isard, and Y. Yu. Steno: Automatic optimization of declarative queries. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 121–131, New York, NY, USA, 2011. ACM.

[25] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. Technical Report UW-CSE-14-05-03, Univeristy of Washington, Apr 2014.

[26] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[27] J. Regehr. Assertions are pessimistic, assumptions are optimistic. http://blog.regehr.org/archives/1096, Feb 2014.

[28] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed SociaLite: A Datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, Sept. 2013.

[29] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 33–40, New York, NY, USA, 2011. ACM.

[30] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[31] S. Viglas, G. M. Bierman, and F. Nagel. Processing declarative queries through generating imperative code in managed runtimes. *IEEE Data Eng. Bull.*, 37(1):12–21, 2014.

[32] B. Wiedermann and W. R. Cook. Extracting queries by static analysis of transparent persistence. *ACM SIGPLAN Notices*, 42(1):199–210, 2007.

[33] B. Wiedermann, A. Ibrahim, and W. R. Cook. Interprocedural query extraction for transparent persistence. In *ACM Sigplan Notices*, volume 43, pages 19–36. ACM, 2008.

[34] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[35] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.

[36] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. Scope: parallel databases meet mapreduce. *The VLDB JournalThe International Journal on Very Large Data Bases*, 21(5):611–636, 2012.