US009792322B2

(12) **United States Patent**
Merriman et al.

(10) **Patent No.: US 9,792,322 B2**
(45) **Date of Patent: *Oct. 17, 2017**

(54) **AGGREGATION FRAMEWORK SYSTEM ARCHITECTURE AND METHOD**

(71) Applicant: **MongoDB, Inc.**, New York, NY (US)

(72) Inventors: **Dwight Merriman**, New York, NY (US); **Eliot Horowitz**, New York, NY (US); **Chris Todd Westin**, Morgan Hill, CA (US)

(73) Assignee: **MongoDB, Inc.**, New York, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 127 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **14/672,901**

(22) Filed: **Mar. 30, 2015**

(65) **Prior Publication Data**

US 2015/0278295 A1 Oct. 1, 2015

Related U.S. Application Data

(63) Continuation of application No. 13/794,710, filed on Mar. 11, 2013, now Pat. No. 8,996,463.
(Continued)

(51) **Int. Cl.**
G06F 17/30 (2006.01)
H04L 1/00 (2006.01)

(52) **U.S. Cl.**
CPC .. ***G06F 17/30412*** (2013.01); ***G06F 17/30283*** (2013.01); ***G06F 17/30442*** (2013.01);
(Continued)

(58) **Field of Classification Search**
CPC G06F 17/30554; G06F 17/3089; G06F 17/30864; H04L 1/0076
(Continued)

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,918,593 A * 4/1990 Huber G06F 17/30595
5,598,559 A * 1/1997 Chaudhuri G06F 17/30454
(Continued)

OTHER PUBLICATIONS

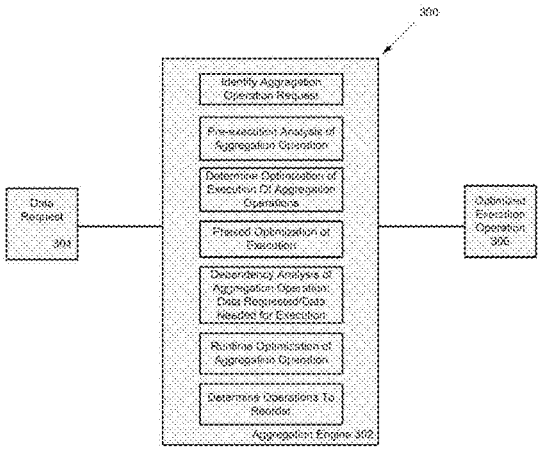
Poder, Tanel, "Oracle Living Books," 2009, http://tech.e2sn.com/oracle/sql/oracle-execution-plan-operation-reference.
(Continued)

Primary Examiner — Frantz Coby
(74) *Attorney, Agent, or Firm* — Wolf, Greenfield & Sacks, P.C.

(57) **ABSTRACT**

Database systems and methods that implement a data aggregation framework are provided. The framework can be configured to optimize aggregate operations over non-relational distributed databases, including, for example, data access, data retrieval, data writes, indexing, etc. Various embodiments are configured to aggregate multiple operations and/or commands, where the results (e.g., database documents and computations) captured from the distributed database are transformed as they pass through an aggregation operation. The aggregation operation can be defined as a pipeline which enables the results from a first operation to be redirected into the input of a subsequent operation, which output can be redirected into further subsequent operations. Computations may also be executed at each stage of the pipeline, where each result at each stage can be evaluated by the computation to return a result. Execution of the pipeline can be optimized based on data dependencies and reordering of the pipeline operations.

27 Claims, 7 Drawing Sheets



Related U.S. Application Data

(60) Provisional application No. 61/676,188, filed on Jul. 26, 2012.

(52) **U.S. Cl.**
CPC .. ***G06F 17/30489*** (2013.01); ***G06F 17/30867*** (2013.01); *G06F 17/3089* (2013.01); *G06F 17/30554* (2013.01); *G06F 17/30864* (2013.01); *H04L 1/0076* (2013.01)

(58) **Field of Classification Search**
USPC 707/637, 713, 715; 709/203, 213, 217, 709/219
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,884,299	A *	3/1999	Ramesh	G06F 17/30445
6,240,406	B1 *	5/2001	Tannen	G06F 17/30436
6,339,770	B1 *	1/2002	Leung	G06F 17/30454
6,385,201	B1	5/2002	Iwata	
6,691,101	B2 *	2/2004	MacNicol	G06F 17/30445 707/714
7,082,473	B2	7/2006	Breitbart et al.	
7,181,460	B2 *	2/2007	Coss	G06F 17/30241
7,467,103	B1	12/2008	Murray et al.	
7,472,117	B2 *	12/2008	Dettinger	G06F 17/30412
7,486,661	B2	2/2009	Van Den Boeck et al.	
7,558,481	B2	7/2009	Jenkins et al.	
7,657,570	B2 *	2/2010	Wang	G06F 17/30286 707/736
7,668,801	B1 *	2/2010	Koudas	G06F 17/30469 707/713
7,957,284	B2	6/2011	Lu et al.	
8,005,868	B2	8/2011	Saborit et al.	
8,037,059	B2	10/2011	Bestgen et al.	
8,082,265	B2	12/2011	Carlson et al.	
8,086,597	B2	12/2011	Balmin et al.	
8,108,443	B2 *	1/2012	Thusoo	G06F 17/30365 707/803
8,170,984	B2 *	5/2012	Bakalash	G06F 17/30457 707/603
8,260,840	B1 *	9/2012	Sirota	G06F 9/5061 370/216
8,296,419	B1 *	10/2012	Khanna	G06F 9/5072 709/201
8,321,558	B1 *	11/2012	Sirota	G06F 9/5011 709/201
8,352,463	B2	1/2013	Nayak	
8,572,031	B2	10/2013	Merriman et al.	
8,589,382	B2	11/2013	Betawadkar-Norwood	
8,712,993	B1 *	4/2014	Ordonez	G06F 17/30489 707/713
8,996,463	B2	3/2015	Merriman et al.	
9,183,254	B1 *	11/2015	Cole	G06F 17/30451
9,262,462	B2	2/2016	Merriman et al.	
9,350,633	B2 *	5/2016	Cudak	H04L 41/0823
2002/0147842	A1	10/2002	Breitbart et al.	
2004/0186817	A1 *	9/2004	Thames	G06F 8/73
2006/0209782	A1	9/2006	Miller et al.	
2008/0002741	A1	1/2008	Maheshwari et al.	
2009/0055350	A1 *	2/2009	Branish, II	G06F 17/30454
2010/0205028	A1	8/2010	Johnson et al.	
2012/0076058	A1	3/2012	Padmanabh et al.	
2012/0254175	A1	10/2012	Horowitz et al.	
2012/0274664	A1	11/2012	Fagnou	
2012/0320914	A1	12/2012	Thyni et al.	
2013/0290471	A1 *	10/2013	Venkatesh	H04L 67/1095 709/214
2013/0339379	A1 *	12/2013	Ferrari	G06F 17/30457 707/766
2014/0032525	A1	1/2014	Merriman et al.	
2014/0032579	A1	1/2014	Merriman et al.	
2014/0032628	A1 *	1/2014	Cudak	H04L 41/0823 709/201
2016/0246861	A1	8/2016	Merriman et al.	

OTHER PUBLICATIONS

Walsh, Norman; Milowski, Alex; Thompson, Henry S., May 11, 2011, "XProc: An XML Pipeline Language," http://www.w3.org/TR/xproc/.

Wikipedia, "Dataflow programming [Oct. 2011 Version]," http://en.wikipedia.org/wiki/Dataflow_programming.

Wikipedia, "Pipeline (Unix) [Sep. 2011 Version]," http://en.wikipedia.org/wiki/Pipeline_(Unix).

* cited by examiner

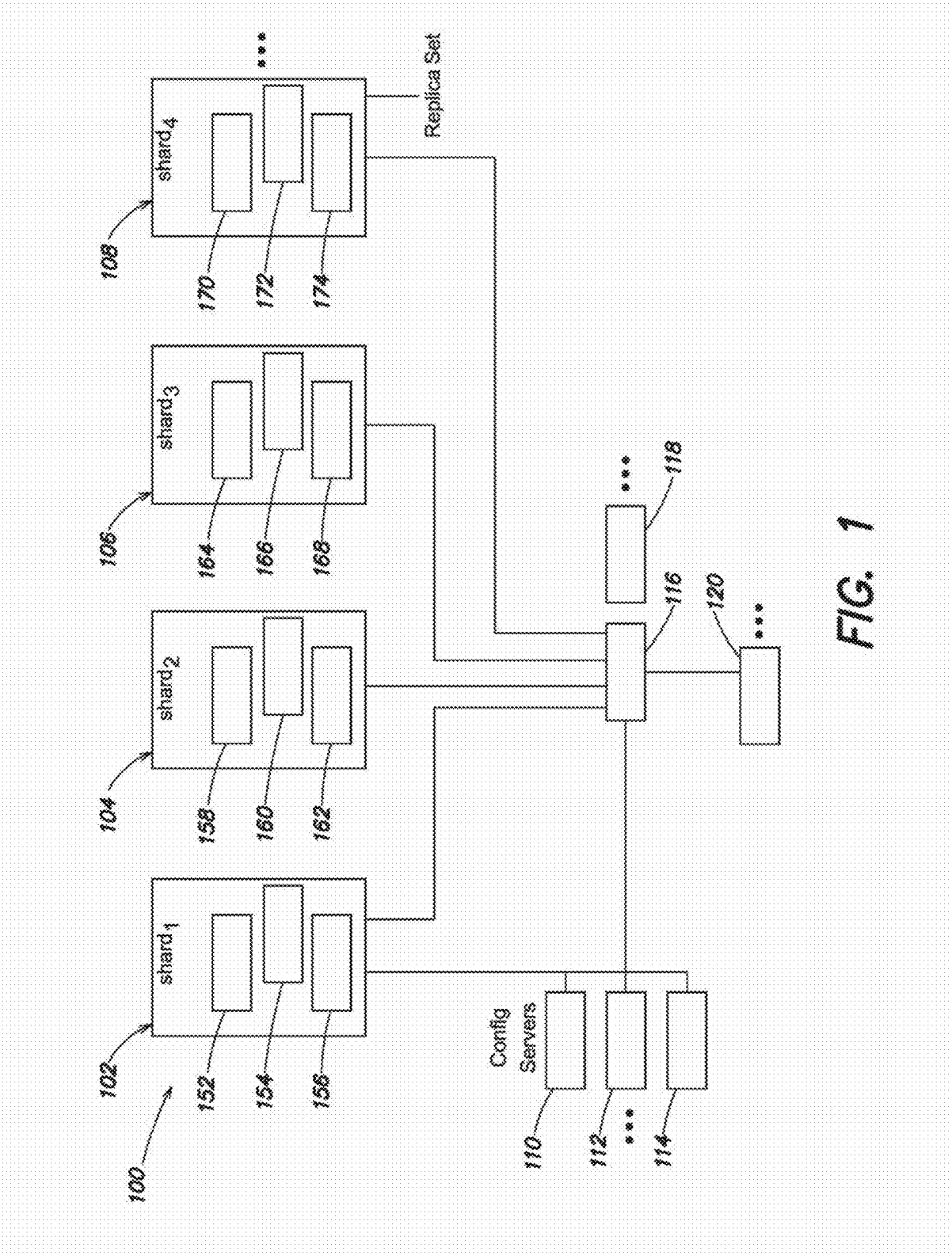

shard1
shard2
shard3
shard4
Replica Set
Config Servers
100
102
104
106
108
110
112
114
116
118
120
152
154
156
158
160
162
164
166
168
170
172
174

FIG. 1

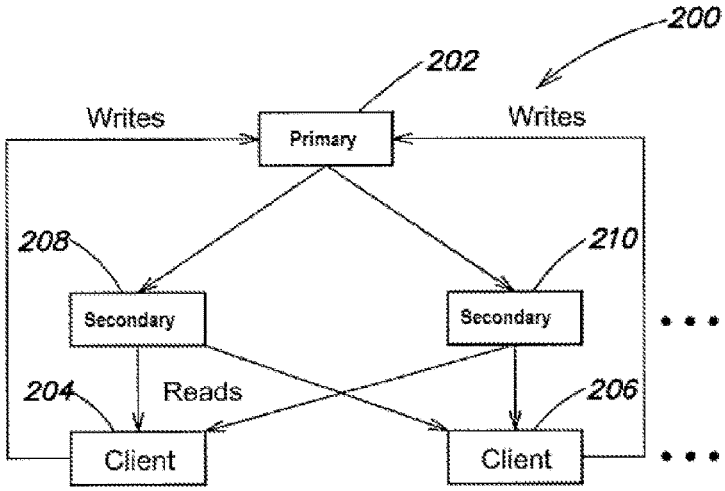

200
202
Writes
Primary
Writes
208
210
Secondary
Secondary
204
Reads
206
Client
Client

FIG. 2

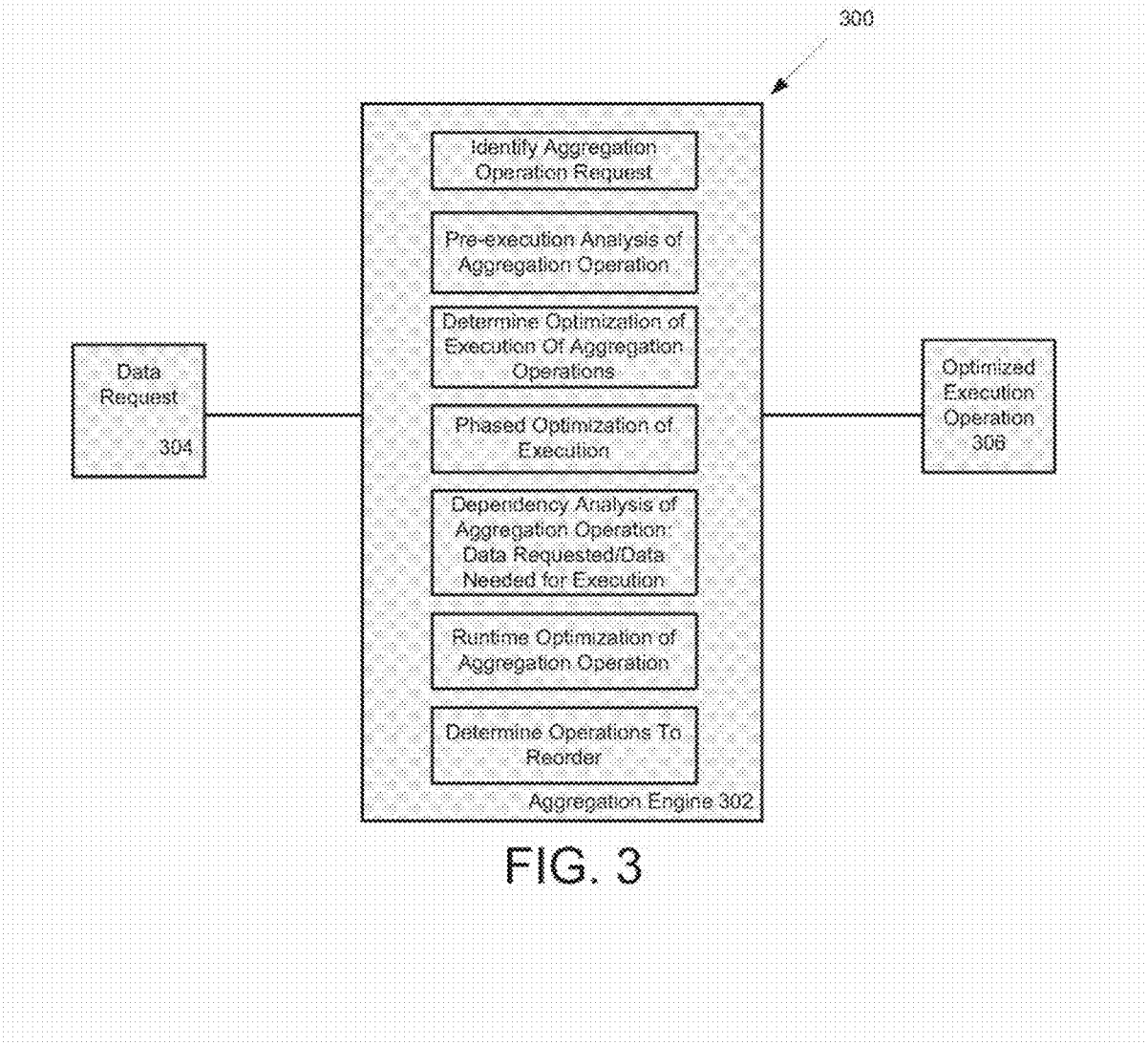

300
Identify Aggregation Operation Request
Pre-execution Analysis of Aggregation Operation
Determine Optimization of Execution Of Aggregation Operations
Phased Optimization of Execution
Dependency Analysis of Aggregation Operation: Data Requested/Data Needed for Execution
Runtime Optimization of Aggregation Operation
Determine Operations To Reorder
Aggregation Engine 302
Data Request 304
Optimized Execution Operation 306

FIG. 3

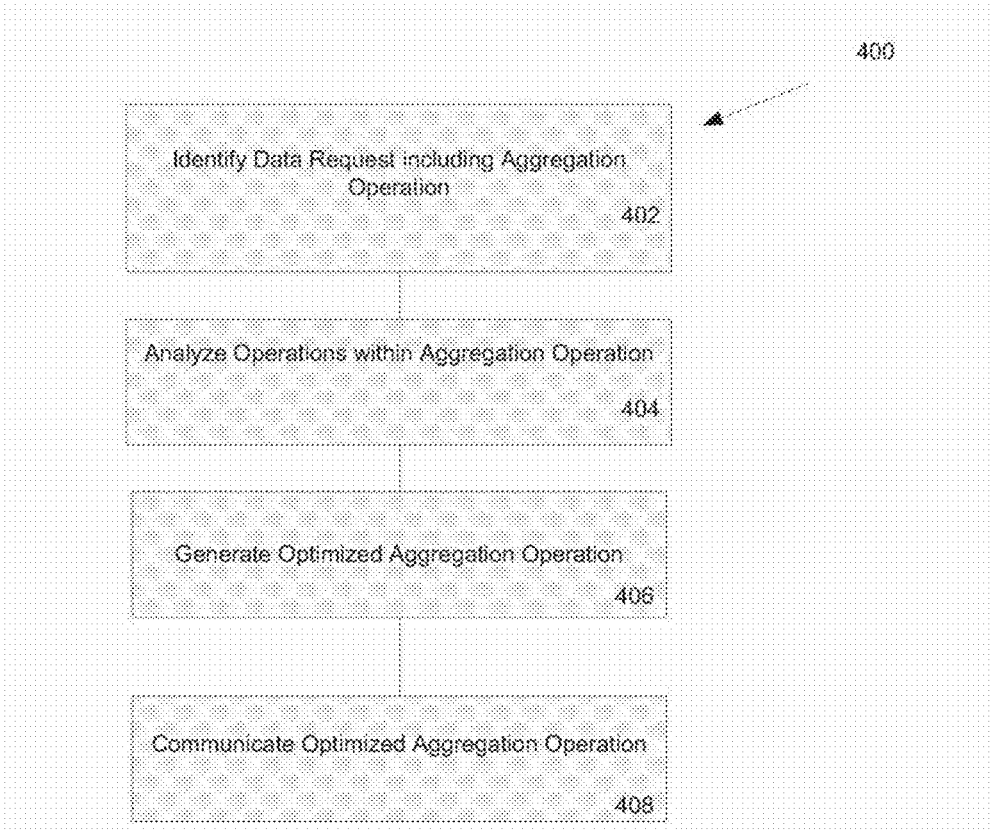

400
Identify Data Request including Aggregation Operation
402
Analyze Operations within Aggregation Operation
404
Generate Optimized Aggregation Operation
406
Communicate Optimized Aggregation Operation
408

FIG. 4

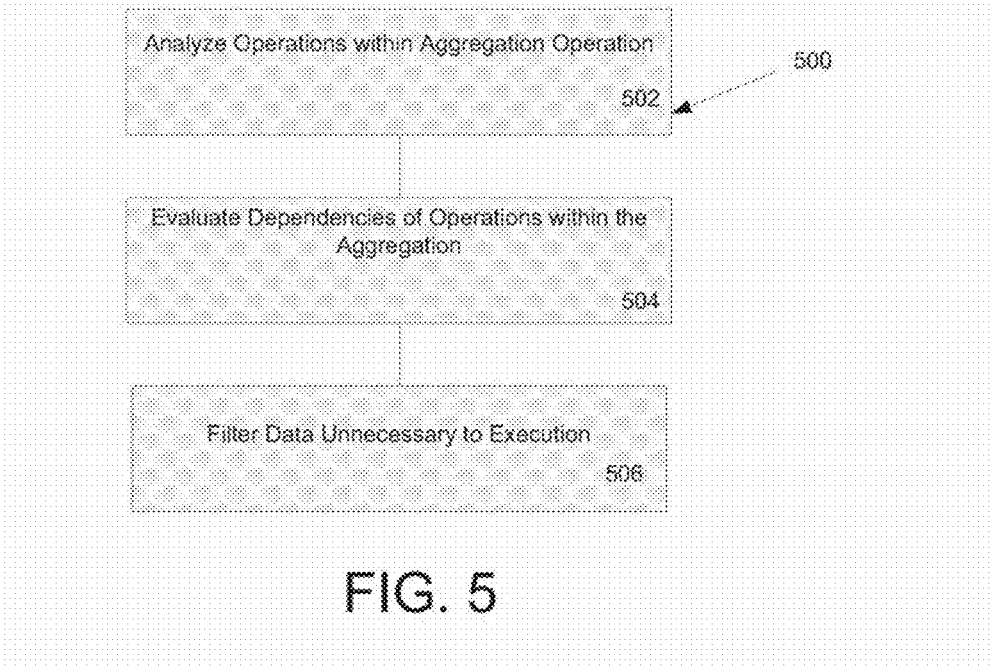
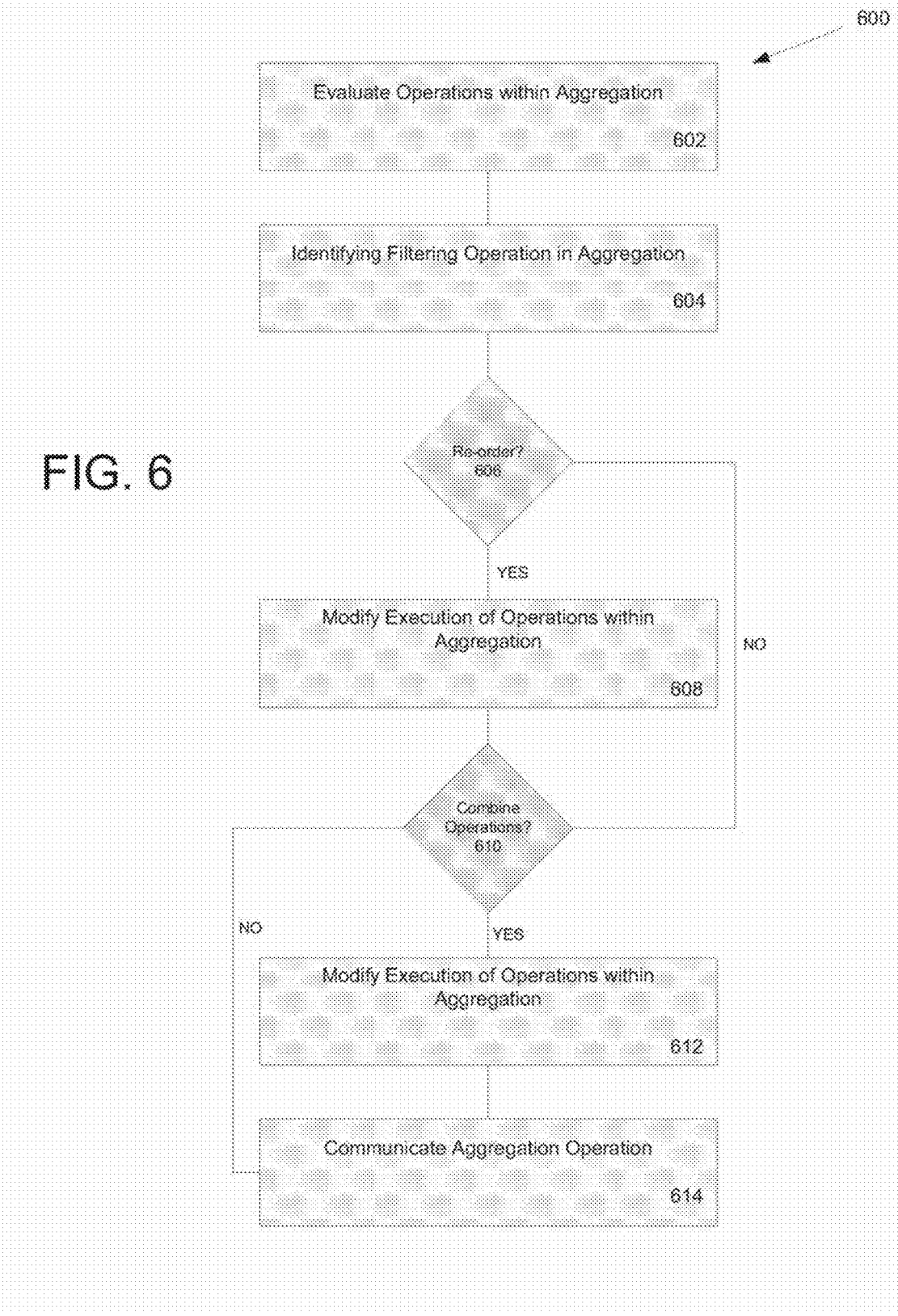

Analyze Operations within Aggregation Operation
502
500
Evaluate Dependencies of Operations within the Aggregation
504
Filter Data Unnecessary to Execution
506

FIG. 5

FIG. 6


600
Evaluate Operations within Aggregation
602
Identifying Filtering Operation in Aggregation
604
Re-order?
606
YES
NO
Modify Execution of Operations within Aggregation
608
Combine Operations?
610
NO
YES
Modify Execution of Operations within Aggregation
612
Communicate Aggregation Operation
614

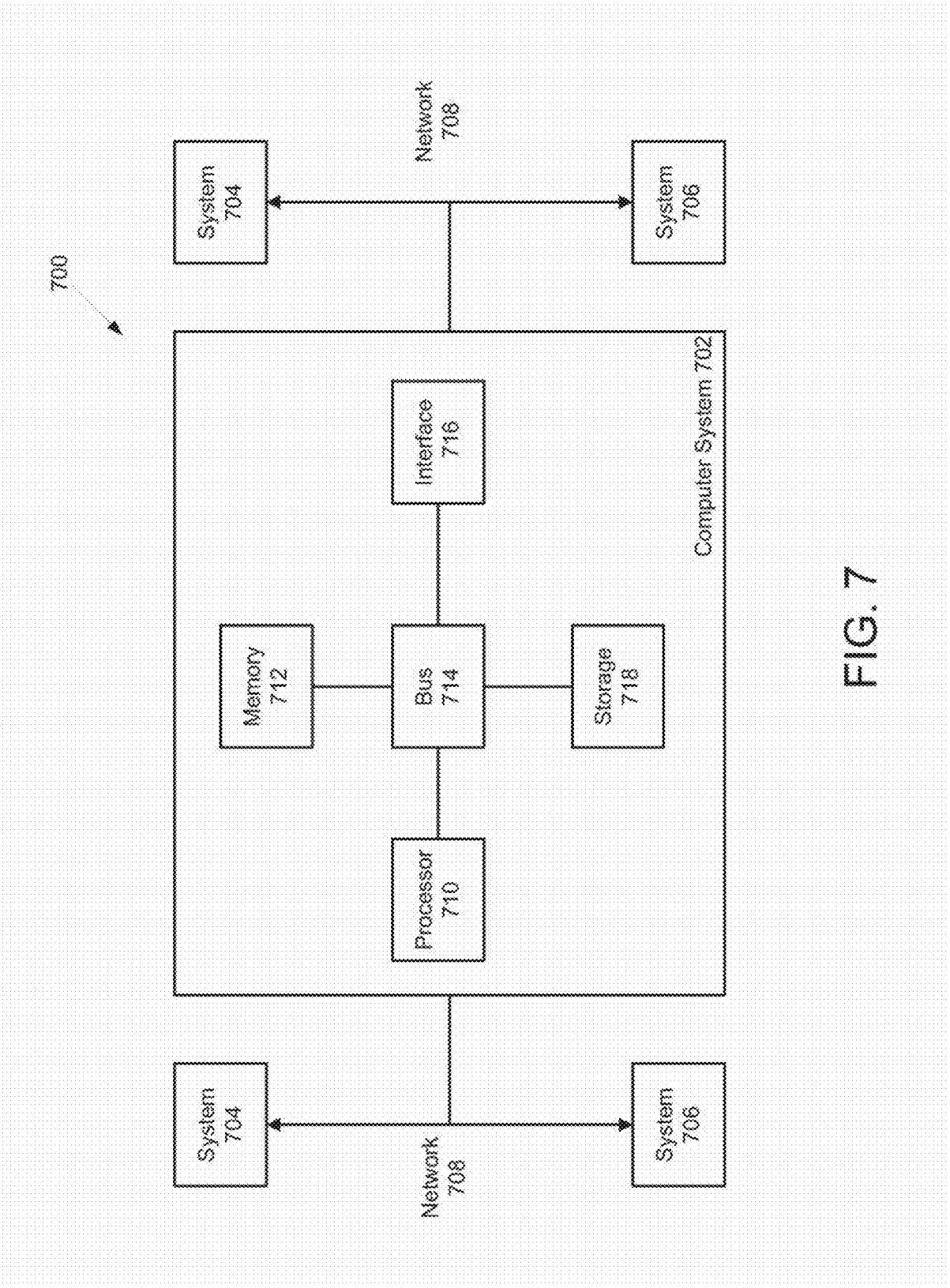

700
System 704
Network 708
System 706
Computer System 702
Interface 716
Memory 712
Bus 714
Storage 718
Processor 710
System 704
Network 708
System 706

FIG. 7

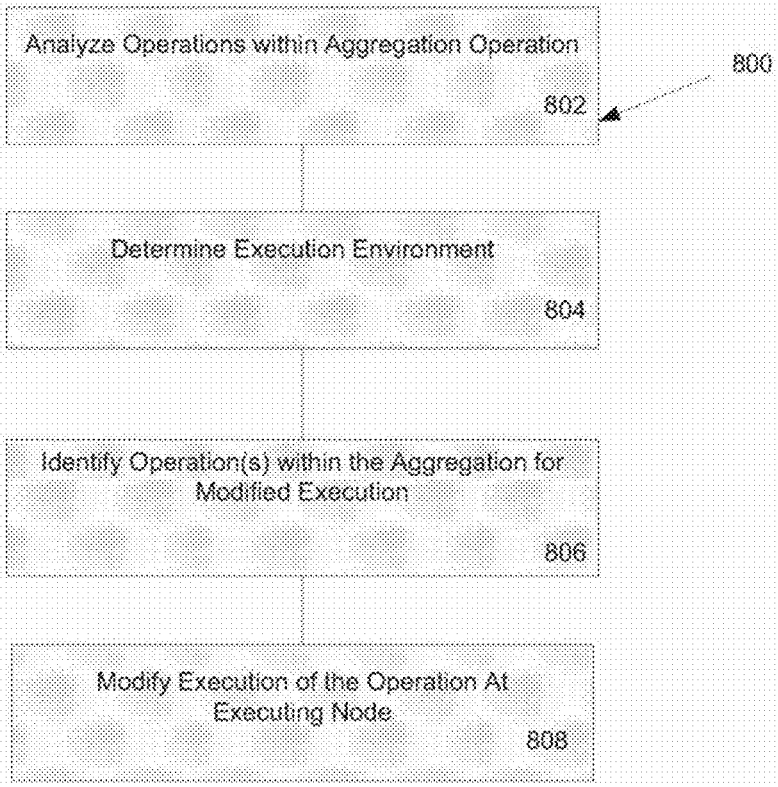

Analyze Operations within Aggregation Operation
802
800
Determine Execution Environment
804
Identify Operation(s) within the Aggregation for Modified Execution
806
Modify Execution of the Operation At Executing Node
808

FIG. 8

AGGREGATION FRAMEWORK SYSTEM ARCHITECTURE AND METHOD

RELATED APPLICATIONS

This application is a continuation of and claims priority under 35 U.S.C. §120 to U.S. Ser. No. 13/794,710, entitled AGGREGATION FRAMEWORK SYSTEM ARCHITECTURE AND METHOD filed on Mar. 11, 2013, which claims priority to U.S. Provisional Application Ser. No. 61/676,188 entitled AGGREGATION FRAMEWORK SYSTEM ARCHITECTURE AND METHOD, filed Jul. 26, 2012, both of which are hereby incorporated by reference in their entirety for all purposes.

BACKGROUND

Speed and consistency of data access are highly important and can be highly variable depending on the architecture of any database environment. The difficulties associated with providing such speed and consistency oftentimes are directly related to the implementation of a specific database, its architecture and/or logical structure. Many conventional databases offerings have moved increasingly away from traditional relational database models and their accompanying constraints to deliver database systems that are more flexible, easier to implement, and highly scalable. Many database systems are now referred to as "NoSQL" databases in response to non-adherence to the widely used relational database management system ("RDBMS") model.

Some example NoSQL database systems can be characterized by not using SQL as a database query language, not providing full ACID guarantees (oftentimes employing eventual consistency in the available data), and can also include distributed, fault-tolerant architecture. The transition away from RMDS models can significantly improve flexibility, scalability, and in some examples tolerance in a database system.

Additional differences exist between relational and non-relational database systems. Including, for example, how data is accessed, how a user requests data, how the data is stored in the underlying database, among other examples. Some conventional non-relational models attempt to support traditional relational data requests like "group by" and other related SQL operations like "join" and self joins. The implementation of such operations in non-relational models can be cumbersome and lead to additional complexity, for example, due to the structure (or its lack of homogeneous structure) of the underlying data in the database.

SUMMARY

It is realized that trade-offs can be required to implement a database system that does not employ traditional "relational" architectures. Some of the sacrifices can include reduced performance trades offs, and can also include increased complexity in querying information from distributed data sources. Accordingly, provided are systems and methods for architecting improved non-relational database models and methods for execution in distributed database systems that can address some of the trade-offs. According to some embodiments, the improved systems and methods include a framework for optimizing sequences or groups of operations submitted, for example, by end users. These sequences of operations can be submitted and executed in aggregate. These aggregate operations can often overwhelm conventional systems due to computational burden, the number of operations, etc. Even simple delay in returning a response to an aggregate operation can frustrate normal use.

According to one aspect, database systems and methods implement a data aggregation framework configured to optimize aggregate operations over non-relational distributed databases, including, for example, data access, data retrieval, data writes, indexing, etc., throughout the distributed database. According to one embodiment, an aggregation architecture is provided that is configured to aggregate multiple operations and/or commands, where the results (e.g., database documents and computations) captured from the distributed database are transformed as they pass through an aggregation operation. The aggregation operation enables the results from a first operation to be redirected into the input of a subsequent operation, which output can be redirected into further subsequent operations. In other embodiments, computations may also be executed at each stage of the aggregate operation, where each result at each stage can be evaluated by the computation to return a result.

According to another aspect, execution of aggregation operations can require significantly more resources than typical data access request. Thus, various embodiments implement execution analysis on the aggregation operations and determine execution optimizations for: reducing at least some of the volume of data needed to complete an aggregation operation, reducing the data communicated between distributed portions of the database, reducing memory requirements for output/pipeline data, re-ordering execution of operations or commands within the aggregation operation, enabling nested pipeline execution on array fields, and parallelizing aggregation operation execution, among other examples.

According to one aspect, a computer implemented method for executing aggregation operations in a non-relational architecture is provided. The method comprises the acts of receiving, at database routing system, database requests from computer executed processes, routing, by the database routing system, the database requests to a plurality of database access managers over a communication network, identifying, by an aggregation engine, an aggregation operation including a plurality of data operations, determining, by the aggregation engine, an optimization for execution of the aggregation operation, modifying, by the aggregation engine, the plurality of data operations to optimize execution, and communicating, by the aggregation engine, the aggregation operation having at least one optimization for execution, and executing, by at least one of the a plurality of database access managers, the modified plurality of data operations to return data from the distributed database.

In one embodiment, the aggregation operation includes the plurality of data operations and a sequence of execution, and the act of modifying, by the aggregation engine, the plurality of data operations to optimize execution includes modifying the sequence of execution of the plurality of data operations. In one embodiment, the method further comprises determining any data dependencies for a group of operations in the plurality of data operations. In one embodiment, the method further comprises an act of storing metadata describing structure of the of data stored in the distributed database. In one embodiment, the method further comprises acts of querying the configuration server to capture the metadata associated with data requested by the plurality of data operations, and generating data dependency information based at least in part in the metadata.

In one embodiment, the method further comprises an act of filtering a request for data defined by the group of operations based on the data dependencies. In one embodi-

ment, the method further comprises an act of identifying a plurality of data fields accessed by the group of operations. In one embodiment, the method further comprises an act of limiting instantiation of any data field not needed for responding to the plurality of operations during execution. In one embodiment, the method further comprises generating data exclusion operations and introducing the data exclusion operations into the plurality of data operations. In one embodiment, the method further comprises an act of optimizing execution of the aggregation operation before routing the operations to the plurality of database access managers. In one embodiment, the method further comprises an act of optimizing execution of the aggregation operation after routing the operations to the plurality of database access managers. In one embodiment, the method further comprises an act of optimizing execution of the aggregation in association with routing and execution of the aggregation operation. In one embodiment, the method further comprises an act of identifying filtering operations with the plurality operations and alter an execution order of the plurality of operations to elevate execution of at least one filtering operation.

In one embodiment, the method further comprises an act of identifying expansion operations with the plurality operations and combine execution of the expansion operation with at least one filtering operation. In one embodiment, the method further comprises managing, by the plurality of database access managers, data access to at least one partition of the data in the distributed database. In one embodiment, the method further comprises an act of organizing the at least one partition of the data according to a base unit of data storage having a non-relational structure. In one embodiment, the base unit of storage includes a document, and the document comprises a data entity including zero or more key/value pairs. In one embodiment, the method further comprises interpreting the aggregation operation for execution, including an order specified group of predefined database operations.

In one embodiment, the method further comprises resolving the predefined database operations into an order specified first phase and at least a second phase. In one embodiment, the method further comprises executing a plurality of rules to evaluate the aggregation operation. In one embodiment, each rule specifies at least one condition and at least one action, and executing the plurality of rules includes identifying an ordering of operations within an aggregation operation that can be modified. In one embodiment, executing the plurality of rules includes generating a modified execution order for the plurality of data operations responsive to an action.

According to another aspect, a distributed database system for optimizing aggregation operations in a non-relational architecture is provided. The system comprises at least one processor operatively connected to a memory, the processor when executing is configured to execute a plurality of system components, wherein the plurality of system components include a router component configured to receive database requests from computer executed processes, and route the database requests to a plurality of database access managers over a communication network, and an aggregation engine configured to identify an aggregation operation including a plurality of data operations, determine an optimization for execution of the aggregation operation, modify the plurality of data operations to optimize execution, and communicate the aggregation operation having at least one optimization, and wherein the plurality of database mangers are configured to execute the modified plurality of data operations to return data from the distributed database.

In one embodiment, the aggregation operation includes the plurality of data operations and a sequence of execution, and wherein the aggregation engine is further configured to modify the sequence of execution of the plurality of data operations. In one embodiment, the aggregation engine is further configured to determine any data dependencies for a group of operations in the plurality of data operations. In one embodiment, the distributed database system further comprises a configuration server includes metadata describing location of data stored in the distributed database. In one embodiment, the aggregation engine is further configured to query the configuration server to capture metadata information associated with the plurality of data operations, and generate data dependency information based at least in part in the metadata information.

In one embodiment, the aggregation engine is further configured to filter a request for data within the group of operations based on the data dependencies. In one embodiment, the aggregation engine is further configured to identify a plurality of data fields accessed by the group of operations. In one embodiment, the aggregation engine is further configured to limit instantiation of any data field not needed for responding to the plurality of operations during execution. In one embodiment, the aggregation engine is further configured to generate data exclusion operations and introduce the data exclusion operations into the plurality of data operations. In one embodiment, the aggregation engine is further configured to optimize execution of the aggregation operation before routing the operations to the plurality of database access managers. In one embodiment, the aggregation engine is further configured to optimize execution of the aggregation operation after routing the operations to the plurality of database access managers.

In one embodiment, the aggregation engine is further configured to optimize execution of the aggregation in association with routing and execution of the aggregation operation. In one embodiment, the aggregation engine is further configured to identify filtering operations with the plurality operations and alter an execution order of the plurality of operations to elevate execution of at least one filtering operation. In one embodiment, the aggregation engine is further configured to identify expansion operations with the plurality operations and combine execution of the expansion operation with at least one filtering operation. In one embodiment, each of the plurality of database access managers manages data access to at least one partition of the data in the distributed database. In one embodiment, the at least one partition of the data comprises data organized according to a base unit of data storage. In one embodiment, the base unit of storage includes a document, and the document comprises a data entity including zero or more key/value pairs.

In one embodiment, the aggregation operation comprises an order specified group of predefined database operations. In one embodiment, at least one of the routing component and the plurality of database access managers are configured to resolve the predefined operations into an order specified first phase and at least a second phase. In one embodiment, the method further comprises a plurality of rules when executed are configured to modify an aggregation operation. In one embodiment, each rule specifies at least one condition and at least one action, wherein the at least one condition identifies an ordering of operations within an aggregation

operation that can be modified. In one embodiment, the at least one action generates a modified execution order for the plurality of data operations.

According to one aspect, a computer implemented method for executing aggregation operations in a non-relational database architecture is provided. The method comprises the acts of receiving, at database routing system, database requests from computer executed processes, routing, by the database routing system, the database requests to a plurality of database access managers over a communication network, identifying, by an aggregation engine, an aggregation operation within the database requests, wherein the aggregation operation includes a plurality of data operations, determining, by the aggregation engine, an optimization for execution of the plurality of data operations, modifying, by the aggregation engine, a default execution of at least one of the plurality of data operations to optimize execution of the at least one of the plurality of data operations, and communicating, by the aggregation engine, the aggregation operation having at least one optimization for execution of the plurality of data operations.

In one embodiment, the aggregation operation includes the plurality of data operations and a sequence of execution, and the act of identifying includes identifying a sequence of execution wherein execution of a first operation in the sequence permits optimization of a subsequent operation. In one embodiment, the act of identifying the sequence of execution includes identifying a result of the first operation generates an ordered set of results used as input by at least one subsequent operation including the second operation. In one embodiment, the second operation includes a group operation, and the act of modifying the default execution of at least one of the plurality of data operations to optimize execution of the at least one of the plurality of data operations includes modifying the execution of the group operation based on analysis of constraints specified by the first operation.

In one embodiment, the constraints imposed include specification of a key for ordering a data stream. In one embodiment, modifying execution of the group operation includes modifying default operation of the group operation to limit instantiation of data structures in executing memory to hold data output of the group operation. In one embodiment, the act of identifying includes identifying a sort operation that generates an ordered set of results upon execution. In one embodiment, the aggregation operation includes the plurality of data operations and a sequence of execution, and the act of identifying includes identifying a sequence of execution wherein execution of an operation in the sequence permits optimization of a preceding operation.

In one embodiment, the operation is configured to filter a set of input data to produce a reduced set of output data, and the act of modifying includes modifying a default operation of the preceding operation based on filter criteria specified in the first operation. In one embodiment, the operation is a limit operation, and the act of modifying comprises modifying default operation of a sort operation within the plurality of data operations to minimize resources instantiated during execution of the sort operation. In one embodiment, the filter criteria specifies a limit on a number of returned results from the database. In one embodiment, the method further comprises an act of determining a data architecture for the data stored on a database. In one embodiment, the modifying, by the aggregation engine, the default execution of at least one of the plurality of data operations to optimize execution of the at least one of the plurality of data operations is responsive to the determined data architecture. In one embodiment, determining the data architecture includes determining the data architecture includes data shards, and the act of modifying includes modifying the default execution of sort operation to sort by merging data results returned from a plurality of the data shards.

According to another aspect, a distributed database system for optimizing aggregation operations in a non-relational architecture is provided. The system comprises at least one processor operatively connected to a memory, the processor when executing is configured to execute a plurality of system components, wherein the plurality of system components include a router component configured to receive database requests from computer executed processes, and route the database requests to a plurality of database access managers over a communication network, an aggregation engine configured to identify an aggregation operation within the database requests, wherein the aggregation operation includes a plurality of data operations, determine an optimization for execution of the plurality of data operations, modify a default execution of at least one of the plurality of data operations to optimize execution of at least one of the plurality of data operations, and communicate the aggregation operation having at least one optimization, and wherein the plurality of database managers are configured to execute the modified operations to return data from the distributed database.

In one embodiment, the aggregation operation includes the plurality of data operations and a sequence of execution, and wherein the aggregation engine is further configured to identify a sequence of execution wherein execution of a first operation in the sequence permits optimization of a subsequent operation. In one embodiment, the aggregation engine is further configured to identify that a result of the first operation generates an ordered set of results used as input by at least one subsequent operation including the second operation. In one embodiment, the second operation includes a group operation, and the aggregation engine is further configured to modify the execution of the group operation based on analysis of constraints specified by the first operation.

In one embodiment, the constraints imposed include specification of a key for ordering a data stream. In one embodiment, the aggregation engine is further configured to modify default operation of the group operation to limit instantiation of data structures in executing memory to hold data output of the group operation. In one embodiment, the aggregation engine is further configured to identify a sort operation that generates an ordered set of results upon execution. In one embodiment, the aggregation operation includes the plurality of data operations and a sequence of execution, and the act of identifying includes identifying a sequence of execution wherein execution of an operation in the sequence permits optimization of a preceding operation. In one embodiment, the operation is configured to filter a set of input data to produce a reduced set of output data, and the aggregation engine is further configured to modify the default operation of the preceding operation based on filter criteria specified in the first operation.

In one embodiment, the operation is a limit operation, and the aggregation engine is further configured to modify default operation of a sort operation within the plurality of data operations to minimize resources instantiated during execution of the sort operation. In one embodiment, the filter criteria specifies a limit on a number of returned results from the database. In one embodiment, the aggregation engine is further configured to determine a data architecture for the data stored on a database. In one embodiment, the aggre-

gation engine is further configured to modify the default execution of at least one of the plurality of data operations responsive to the determined data architecture. In one embodiment, the act of determining the data architecture includes an act of determining the data architecture includes data shards, and the aggregation engine is further configured to modify the default execution of a sort operation to sort by merging data results returned from a plurality of the data shards.

According to one aspect, a computer implemented method for executing aggregation operations in a non-relational database architecture is provided. The method comprising the acts of receiving, at database routing system, database requests from computer executed processes, routing, by the database routing system, the database requests to a plurality of database access managers over a communication network, identifying, by an aggregation engine, an aggregation operation within the database requests, wherein the aggregation operation includes a plurality of data operations, determining, by the aggregation engine, an optimization for execution of the aggregation operation, wherein the act of determining the optimization for execution includes an act of managing memory allocations during execution of the aggregation operation, communicating, by the aggregation engine, the aggregation operation having at least one optimization for execution of the plurality of data operations.

In one embodiment, the act of managing the memory allocations during execution of the aggregation operation includes an act of generating an string table for data field names associating a field value with the data field names returned by execution of the aggregation operation. In one embodiment, the method further comprises an act of replacing the data field names with a respective field value. In one embodiment, the method further comprises an act of communicating data and the respective field value as input to a subsequent operation in the aggregation operation. In one embodiment, the act of determining, by the aggregation engine, the optimization for execution of the aggregation operation includes an act of performing execution analysis of the aggregation operation to determine data accessed by the plurality of operations.

In one embodiment, the act of managing memory allocations during execution includes an act of identifying at least one of the plurality of data operations targeting index key data. In one embodiment, the method further comprises an act of separating execution of the aggregation operation into phases including at least a index data phase. In one embodiment, the method further comprises an act of releasing execution resources assigned to operations within the aggregation operation upon communication of data to a subsequent operation. In one embodiment, the method further comprises an act of analyzing the plurality of operations to identify cumulative operations. In one embodiment, the method further comprises an act of freeing memory resources employed by the cumulative operations upon emitting their results.

According to another aspect, a distributed database system for optimizing aggregation operations in a non-relational database architecture is provided. The system comprises at least one processor operatively connected to a memory, the processor when executing is configured to execute a plurality of system components, wherein the plurality of system components include a router component configured to receive database requests from computer executed processes, and route the database requests to a plurality of database access managers over a communication network, an aggregation engine configured to identify an aggregation operation within the database requests, wherein the aggregation operation includes a plurality of data operations, determine an optimization for execution of the aggregation operation by managing memory allocations during execution of the aggregation operation, communicate the aggregation operation having at least one optimization, and wherein the plurality of database managers are configured to execute the modified operations to return data from the distributed database.

In one embodiment, the aggregation engine is further configured to generate an string table for data field names associating a field value with the data field names returned by execution of the aggregation operation. In one embodiment, the aggregation engine is further configured to replace the data field names with a respective field value. In one embodiment, the aggregation engine is further configured to communicate data and the respective field value as input to a subsequent operation during execution of the aggregation operation. In one embodiment, the aggregation engine is further configured to determine data accessed by the plurality of operations prior to execution. In one embodiment, the aggregation engine is further configured to identify at least one of the plurality of data operations targeting index key data.

In one embodiment, the aggregation engine is further configured to separate execution of the plurality of data operations into phases including at least an index data phase for execution of at least one of the plurality of data operations targeting index key data. In one embodiment, the aggregation engine is further configured to release execution resources assigned to operations within the aggregation operation upon communication of data to a subsequent operation. In one embodiment, the aggregation engine is further configured to analyze the plurality of operations to identify cumulative operations. In one embodiment, the aggregation engine is further configured to free memory resources employed by the cumulative operations upon emitting their results.

According to one aspect, a computer implemented method for executing aggregation operations in a non-relational database architecture is provided. The method comprises the acts of receiving, at database routing system, database requests from computer executed processes, routing, by the database routing system, the database requests to a plurality of database access managers over a communication network, identifying, by an aggregation engine, an aggregation operation within the database requests, wherein the aggregation operation includes a plurality of data operations, analyzing, by the aggregation engine, the plurality of data operations, and identifying, by the aggregation engine, a group of operations from the plurality of data operations configured to generate a data processing window, separating, by the aggregation engine, the aggregation operation into phases of operation including at least a phase configured to execute the group of operations that generate the data processing window upon execution.

In one embodiment, the method includes an act of caching an input data stream being processed by the group of operations that generate the data processing window. In one embodiment, the method further comprises an act of re-executing the group of operations against a newly defined window. In one embodiment, the method further comprises an act of inputting the cached data for processing of the newly defined window. In one embodiment, the aggregation engine is further configured to identify a skip and limit operation as at least part of the group of operations.

According to another aspect, a computer implemented method for executing aggregation operations in a non-relational database architecture is provided. The method comprises the acts of receiving, at database routing system, database requests from computer executed processes, routing, by the database routing system, the database requests to a plurality of database access managers over a communication network, identifying, by an aggregation engine, an aggregation operation within the database requests, wherein the aggregation operation includes a plurality of data operations, analyzing, by the aggregation engine, the plurality of data operations, and identifying, by the aggregation engine, a group of operations configured to execute a subordinate aggregation operation within the plurality of data operations, separating, by the aggregation engine, the aggregation operation into phases of operation including at least a phase configured to execute the group of operations configured to execute the sub-ordinate aggregation operation within the plurality of operations.

According to another aspect, a distributed database system for optimizing aggregation operations in a non-relational database architecture is provided. The system comprises at least one processor operatively connected to a memory, the processor when executing is configured to execute a plurality of system components, wherein the plurality of system components include a router component configured to receive database requests from computer executed processes, and route the database requests to a plurality of database access managers over a communication network, an aggregation engine configured to identify an aggregation operation within the database requests, wherein the aggregation operation includes a plurality of data operations, analyze the plurality of data operations, and identify a group of operations from the plurality of data operations configured to generate a data processing window, separating the aggregation operation into phases of operation including at least a phase configured to execute the group of operations that generate the data processing window upon execution.

Still other aspects, embodiments, and advantages of these exemplary aspects and embodiments, are discussed in detail below. Any embodiment disclosed herein may be combined with any other embodiment in any manner consistent with at least one of the objects, aims, and needs disclosed herein, and references to "an embodiment," "some embodiments," "an alternate embodiment," "various embodiments," "one embodiment" or the like are not necessarily mutually exclusive and are intended to indicate that a particular feature, structure, or characteristic described in connection with the embodiment may be included in at least one embodiment. The appearances of such terms herein are not necessarily all referring to the same embodiment. The accompanying drawings are included to provide illustration and a further understanding of the various aspects and embodiments, and are incorporated in and constitute a part of this specification. The drawings, together with the remainder of the specification, serve to explain principles and operations of the described and claimed aspects and embodiments.

BRIEF DESCRIPTION OF THE DRAWINGS

Various aspects of at least one embodiment are discussed below with reference to the accompanying figures, which are not intended to be drawn to scale. Where technical features in the figures, detailed description or any claim are followed by reference signs, the reference signs have been included for the sole purpose of increasing the intelligibility of the figures, detailed description, and claims. Accordingly, neither the reference signs nor their absence is intended to have any limiting effect on the scope of any claim elements. In the figures, each identical or nearly identical component that is illustrated in various figures is represented by a like numeral. For purposes of clarity, not every component may be labeled in every figure. The figures are provided for the purposes of illustration and explanation and are not intended as a definition of the limits of the invention. In the figures:

FIG. **1** is a block diagram of an example system architecture implementing an aggregation framework according to one embodiment;

FIG. **2** is a block diagram of an example system architecture of a replica set according to one embodiment;

FIG. **3** is a block diagram of an example system for identifying and executing aggregation operations on distributed data, according to one embodiment;

FIG. **4** is an example process flow for optimizing execution of an aggregation operation, according to one embodiment;

FIG. **5** is an example process flow for analyzing operations within an aggregation operation, according to one embodiment;

FIG. **6** is an example process flow for analyzing operations within an aggregation operation, according to one embodiment;

FIG. **7** is a block diagram of a computer system on which various embodiments of the invention may be practiced; and

FIG. **8** is an example process flow for analyzing operations within an aggregation operation, according to one embodiment.

DETAILED DESCRIPTION

As described above, conventional architectures of non-relational database systems oftentimes cannot implement relational like requests that rely on the underlying structure of the data in a relational database. Typically, other operations that perform similar functionality are implemented; however, the ease of use and robustness of those operations with similar functionality can be inconsistent, especially where multiple operations are required. Further, aggregation of operations to achieve similar functionality can result in computational burden and/or over utilization of resources. Over utilization of resources can result in performance issues across an entire database. Accordingly, there is a need for database systems and methods that enable submission of aggregation operations and further need for systems and methods that analyze and optimize execution of aggregate operations to insure greater performance throughout, for example, a distributed database.

According to one aspect, database systems and methods implement a data aggregation framework configured to optimize aggregate operations, including those that mirror relational operations over non-relational distributed databases, including, for example, data access, data retrieval, data writes, indexing, etc., throughout a distributed database. In one embodiment, the aggregation framework can be managed by an aggregation engine. Shown in FIG. **3** is an embodiment of an example aggregation engine **302** on database system **300**.

Data requests (e.g., **304**) can be received from clients, users, applications, and/or any other processing entity capable of connecting and communicating with system **300** and a database management system. In some embodiments system **300** can be implemented as a stand alone computer system (e.g., computer system **700**, FIG. **7**). In other embodiments, system **300** can be integrated and/or opera-

tively connected to a database management system (e.g., **100**, FIG. **1**). In further embodiments, system **300** can be a component of a database management system (e.g., **100**, FIG. **1**) and be executed in conjunction with other components and/or processes for managing a distributed database system. Upon receipt of a data request **304**, an aggregation engine **302** can be configured to identify the data request as comprising and/or including an aggregation operation.

In some implementations, identification of the aggregation operation can occur using different processes, for example, a routing process configured to route data requests within a distributed database system, discussed in greater detail below (See. FIG. **1**, **116-118**). For example system **300** can include a routing process that can be further configured to identify and to communicate an aggregation operation to the aggregation engine **302**. In some embodiments, the routing processing can trigger the aggregation engine in response to identifying an aggregation operation. In other embodiments, an aggregation engine can be executed as part of a routing process. In still further embodiments, one or more aggregation engines can be instantiated on systems throughout a distributed database configured to manage distributed execution of aggregation operations and/or distributed portions of aggregation operations.

According to one embodiment, once the aggregation engine **302** has identified or received an aggregation operation, the engine is configured to parse the operation to determine if any optimization of the aggregation operation is appropriate. For example, the aggregation engine can be configured to parse a group of operations within an aggregation wrapper to identify any filtering operations. Filtering operations identified by the aggregation engine **302** include, for example, any operation(s) that when executed reduces the set of data being analyzed, accessed, or transformed during the execution of the aggregation operation.

In one example, by re-ordering the operations contained within an aggregation wrapper filtering operations can be executed first or at an earlier position within the aggregation pipeline. Earlier execution of filtering operations can be configured to reduce the amount of data accessed, and potentially the number of distributed systems touched to respond to the aggregation operations can also be significantly reduced. In some executions, re-ordering can also be performed at each of the distributed systems that contain data associated with an aggregation operation. The re-ordered aggregation operation can then be executed on the distributed database as an optimized version, e.g., **306**, of the original aggregation operations.

For example, an aggregation operation can require access to a plurality of database instances to resolve the data access requests contained within the aggregation operation. According to one embodiment, the plurality of database instances (and the systems that host them) can be identified by a routing process on the distributed database. The aggregation operations can be evaluated for optimization prior to distribution to the plurality of database instances, and the optimized operation can be communicated to the plurality of database instances. Where a global optimization is not possible and even where global optimization has been identified, further optimization can take place at the systems hosting the plurality of database instances to generate an optimized aggregation operation **306**.

In some embodiments, an aggregation engine can be distributed across the servers and/or systems of the distributed database. The distributed aggregation engine can further optimize execution of the operations within an aggregation wrapper, for example, at the system being accessed to provide data responsive to the aggregation operation. In other embodiments, a plurality of aggregation engines can be executed at each system hosting one or more database instances. In further examples, servers hosting routing processes can also execute one or more aggregation engines for optimizing execution of data request. In one embodiment, an aggregation engine can be executed as part of and/or in conjunction with a routing process.

In some implementations, the aggregation engine performs pre-execution analysis of an aggregation operation and any operations defined therein to determine if re-ordering execution can reduce the computational burden of the execution of the aggregation operation. Further implementations include analysis of the operations in the aggregation wrapper to determine if substitutions or elimination of data requested, documents, or data fields can be made within the operations in the aggregation wrapper. Simple substitutions can include, for example, instances where an output field of an operation is equal to an input field. Data fields, data documents, and/or operations can also be eliminated if for example, the documents, data fields and/or operations are not carried through the entire aggregation pipeline. In another example, only data fields actually required for output need to be processed through execution of an aggregation operation.

According to one embodiment, dependency analysis for the aggregation operation and/or data needed for the operations in the aggregation operations is executed by the aggregation engine. The dependency analysis enables the aggregation engine to determine if, for example, data substitutions are appropriate, if data does not need to be accessed to complete an operation, if data accesses can be eliminated, etc. In one embodiment, the aggregation engine is configured to perform a dependency graph analysis of the any one or more of the operations requested in the aggregation operation, the data accesses required by the aggregation operation, and any data transformations required by the aggregation operation to complete execution of the aggregation operations. In one example, the dependencies of the operations and/or data can be graphically represented as a tree structure. The branches with no connection to an output, to an input of a subsequent operation(s), or branches that are filtered out by subsequent operations, can be eliminated from preceding executions of operations. For example, in a logical model of dependencies, leaf nodes in a tree structure of dependencies can be pruned by the aggregation engine if the leaf node has no connection to subsequent stages or an output. In another example, dependency analysis enables the aggregation operation to identify when a filtering operation can take place prior to its listed execution order.

In some implementations, the execution of aggregation operations can be separated into stages, including one or more transformation stages, where data in the database is accessed and operated on, and the results are used as input in a subsequent operation or set of operations. The results of one or more transformation stages can then be used during a projection stage where final results are accumulated and projected into an output for communication to, for example, a requesting entity. To extend the UNIX|metaphor, a first transformation stage can include a first operation that defines the data being accessed, the result of which is used as an input to a second transformation stage, to filter the data accessed, which can be passed to a third transformation stage that operates on the filtered data, and a projection stage can then deliver final results of the transformations as, for example, an accumulation of the transformation stages,

executed on a plurality of servers, hosting a plurality of database instances. Each of the preceding stages can include optimizations identified locally at each database instance by, for example, a local aggregation engine, and further, the entire execution can include optimizations to streamline processing of the preceding stages.

In some embodiments, certain operations can be readily optimized. For example, filtering operations by their nature typically filter or reduce a set of input data when generating a set of output data. Match$ is one example of a filter operation. Match$ can be configured to return documents and/or data fields within stored documents that have the same criteria defined in the Match$ operation. In other words, the Match$ operation filters a set of inputs to generate a set of output data that can include documents, data fields within documents, and/or values of data fields, among other examples. By moving such filtering operations into an earlier execution position in a sequence of operations (e.g., an aggregation pipeline), an aggregation engine can deliver improved performance over all subsequent operations in the sequence.

Other operations may improve performance by later execution. In particular, Unwind$ is an example operation configured to flatten out data records in a document. In some embodiments, BSON type documents can include references to other BSON documents. In essence, documents as a unit of storage of data can support nesting of documents within other documents. When an operation is executed to flatten a set of documents, the computational burden can be large. Limiting the volume of data that is unwound serves to improve performance of the operations. Thus, in some examples, demoting the execution of an unwind operation can reduce the computation associated with an aggregation operations by, for example, allowing prior operations to limit the set of documents being passed to the unwind operation. In some evaluations, the number of operations can increase if re-order, thus, the aggregation engine can be configured to identify if re-ordering of an Unwind$ operations results in improved performance and/or a reduction in the data operated on.

According to one aspect, provided is an aggregation framework which can include a set of defined operations for execution on a non-relational distributed database, an aggregation engine associated with routing operations in the non-relational distributed database for identifying and optimizing a sequence of operations, and a standard format for submitting requests for aggregation operations to be executed on the database, among other options. The aggregation framework can be implemented on a variety of database architectures. For example, an aggregation framework and/or an aggregation engine can be integrated into the database architectures shown in FIGS. **1** and **2**.

In some embodiments, the aggregation framework is implemented on database systems and through database methods that can improve data access speeds, reduce data resources consumed during read/writes, and simplify end-user data access, and/or query language. In another embodiment, an aggregation framework is implemented on a distributed database system that supports eventually consistency and fault tolerance through replica sets. Replica sets can be implemented as a group of nodes that shares responsibility for a portion of data in a database (e.g., FIG. **2**). In some embodiments, each replica set is configured to host one or more shards or partitions of data. Co-pending application Ser. No. 13/078,104, entitled "SYSTEM AND METHOD FOR OPTIMIZING DATA MIGRATION IN A PARTITIONED DATABASE," filed on Apr. 2, 2011 and incorporated herein in its entirety, discusses some embodiments of shared databases and/or shard clusters on which various aspects can be practiced.

Each partition can be implemented as one or more shards of data. Configuration servers can also be implemented to maintain configuration data across a plurality of shards and/or replica sets. The configuration data can reflect, for example, what data is stored in each of the shards. In some implementations, a database router can be configured to accept requests for database access, and route data operations based on the configuration data. Various database environments (e.g., router, config servers, and shard servers) can support various data architectures. In one embodiment, the base unit of data storage is configured as a document.

Some examples of document organization formats include the known JSON (JavaScript Object Notation) and can also include BSON (binary encoded serialization of JSON) formatting for documents. BSON is a binary format in which zero or more key/value pairs are stored as a single entity. The BSON entity can be referred to as a document. In some examples, BSON is designed to be efficient in space, but in many cases is not much more efficient than JSON. In some cases, BSON can employ more space than JSON to encode information. In one embodiment, this results from one of the BSON design goals: traversability. In some examples, BSON adds some additional information to documents, like length prefixes, that make the document easier and faster to traverse. In some embodiments, encoding data to BSON and decoding from BSON can be performed very quickly in most languages due to the use of C data types. BSON documents can also be specifically designed to be fast to encode and decode. For example, integers are stored as 32 (or 64) bit integers, so they do not need to be parsed to and from text. This uses more space than JSON for small integers, but is much faster to parse. BSON documents can also reference other BSON documents. A BSON document can reference a plurality of other BSON documents. In some examples, further references can be made by the referenced BSON document(s).

According to one environment of a database management system, one or more servers can host multiple shards of data, and each shard can be configured to respond to database requests as if the shard was a complete database. In one embodiment, a routing process can be employed to insure the database requests are routed to the appropriate shard or shards. "Sharding" refers to the process of partitioning the database into partitions, which can be referred to as "shards." FIG. **1** shows a block diagram of an example architecture for a database system **100**, on which the aggregation framework can be implemented. The database system **100** has been specially configured as to provide access to stored data through database shards. The database can be hosted on a plurality of servers hosting a plurality of shards. The database system can be identified as a shard cluster, that is the grouping of shards that collectively represent the data within the database. A shard cluster typically comprises multiple shard servers (e.g., **102-108**) hosting multiple partitions (e.g., **152-174**) or shards of data, one or more configuration servers (e.g., **110-114**) for metadata management, and shard router processes (e.g., **116-118**) for directing data access requests, based on, for example, metadata on the configuration servers. Metadata for the shard cluster can include, for example, information on the ranges of data stored in each partition, information associated with managing the shard cluster, partition counts, number of shard servers, data index information, partition size constraints, data distribution thresholds, among other options.

Each shard of data (e.g., **152-174**) can be configured to reside on one or more servers executing database operations for storing, retrieving, managing, and/or updating data. In some embodiments, a shard server **102** contains multiple partitions of data, which can also be referred to as "chunks" of database data. In some embodiments, a shard of data corresponds to a chunk of data. A chunk is also a reference to a partition of database data. A shard or chunk can be configured as a contiguous range of data from a particular collection in the database. Collections are logical organizations of subsets of database data. In one example, a collection is a named grouping of the data, for example, a named grouping of documents. As discussed above, documents can be as a base unit of storage of data the database. Some examples of document organization formats include the known JSON (JavaScript Object Notation) and BSON (binary encoded serialization of JSON) formatting for documents.

For example, documents can be organized into a collection based on a named grouping, where the named grouping is homogenous or heterogeneous. In some embodiments, collections are organizations of database data similar to relational database tables.

According to one embodiment, configurations within a shard cluster can be defined by metadata associated with the database referred to as shard metadata. Shard metadata can include information on collections within a given database, the number of collections, data associated with accessing the collections, database key properties for a given collection, ranges of key values associated with a given partition, shard, and/or chunk of data within a given collections, to provide some examples.

The three dots illustrated next to the system components, in FIG. **1**, indicate that the system component can be repeated. In some embodiments, adding additional shards, configuration servers, and/or shard routing processes can increase the capacity of the database system. The shard router processes **116-118** handle incoming requests from clients **120** (e.g., applications, web services, user initiated requests, application protocol interfaces, etc). The router processes **116-118** are configured to provide a transparent interface to handle database requests. In particular, client **120** need not know that a database request is being served by a sharded database. The shard router processes receive such client requests and route the database requests to the appropriate shard(s), e.g., **152-174** on shard servers **102-108**.

According to some embodiments, a router process, e.g., **116**, can be configured to operate as a routing and coordination process that makes the various components of the cluster look like a single system, for example, to client **120**. In response to receiving a client request, the router process **116** routes the request to the appropriate shard or shards. In one embodiment, the router process (e.g., **116** or **118**) is configured to identify aggregation operations, analyze the operations within an aggregation wrapper to determine what data is necessary for a given operation and route requests to the shards in which that data is stored.

In some embodiments, the router process can be configured to execute an aggregation engine that identifies and optimizes execution of the aggregation operations. In one embodiment, the aggregation engine (e.g., aggregation engine **302**, FIG. **3**) can be configured to pre-execute the various operations in the aggregation wrapper to establish dependency graphs of the operation results and data. Using the dependency graphs the aggregation engine can define what data is necessary for completing the aggregation operation and further provide that analysis to routing processes for requesting the data from appropriate shards.

The shard(s) return any results to the router process. The router process **116** can merge any results and communicate the merged result back to the client **120**. In some examples, the router process **116** is also configured to establish current state information for the data distributed throughout the database by requesting metadata information on the database from the configuration server(s) **110-114**. In one example, the request for metadata information can be executed on startup of a routing process. Further requests can be initiated by the routing process and/or can be initiated by a configuration server. In another example, a change at the configuration server can trigger a distribution of updates to any routing processes.

In some embodiments, any changes that occur on the configuration server(s) can be propagated to each router process **116-118**, as needed. In one example, router processes **116-118** can be configured to poll the configuration servers(s) **110-114** to update their state information periodically. In others examples, router processes can be configured to poll the configuration servers(s) **110-114** to update their state information on a schedule, periodically, intermittently, and can be further configured to received updates pushed from the configuration server(s) **110-114** and/or any combination of thereof.

According to some further embodiments, router processes can run on any server within the database and/or on any number of server(s) that is desired. For example, the router processes can be executed on stand-alone systems, and in other examples, the router processes can be run on the shard servers themselves. In yet other examples, the router processes can be run on application servers associated with the database.

According to one embodiment, configuration server(s) **110-114** are configured to store and manage the database's metadata. In some examples, the metadata includes basic information on each shard in the shard cluster (including, for example, network communication information), server information, number of chunks of data, chunk version, number of shards of data, shard version, and other management information for routing processes, database management processes, chunk splitting processes, etc. According to some embodiments, shard or chunk information can be the primary data stored by the configuration server(s) **110-116**. In some examples, shard and/or chunks are defined by a triple (collection, minKey, and maxKey) and the metadata stored on the configuration servers establishes the relevant values for a given chunk of data.

According to another embodiment, aggregation of data access operations across a plurality of shards of data and/or replica sets can be provided to reduce the complexity associated with generating the data request and/or to streamline computer execution of capturing the requested data. In some embodiments, aggregation operations can include pipeline and expressions operations. Pipeline operations are conceptually similar to the UNIX-like command line pipe operator "|". In UNIX environments, the pipe operator connects a series of operations/commands where the first command in the list is executed, the result is used as the input into the next data request operations, where the first command in the list is executed, and the result is used as the input into the next operation. Each output is directed into the next operation as input until a final result is generated.

Expressions calculate values from the accessed data, as the data is processed through the pipeline. The expression can collect these values and further calculate other values

from other data processed through the pipeline. In one environment, the aggregation operation is executed against data stored as documents in the database. The expression can collect values from the documents that have flowed through the pipeline.

For example, end-users can be provided a user interface through the system, in which the users are able to define a series of operations inside an aggregation wrapper. The aggregation wrapper is identified by the system using a tag, key word, and/or name that identified the type of operation to the system. For example, a user may enter in a command line: "db.article.aggregate {command [options] syntax; command2 [options] syntax; command3 [options] syntax]}" which reflects an example aggregation wrapper and example commands within the wrapper to be executed. Each command can have its own syntax (e.g., file names, matching criteria) and options specified that filter and/or augment the execution and results of the command and associated syntax. The aggregation operation when executed performs the series of operations to capture or compute data from the distributed database as a collective or sequential execution. Additionally, various embodiments can analyze and reduce the computational complexity in executing aggregation operations across a distributed database. In some embodiments, the nature of the underlying data can influence how aggregation operations are executed and/or optimized for specific data.

In some embodiments, a shard cluster also includes processes for automatic failover and/or recovery. Proper operation of a shard cluster can require that each shard always remain online, or from a more practical standpoint, as available as is reasonably possible. Inconsistent results can be returned if one of the shards hosting necessary data is unavailable. According to one embodiment, each shard server in a shard cluster can be implemented as a replica set, e.g., shard server **108**. A replica set can be configured to perform asynchronous replication across a series of nodes, with various processes implemented to handle recovery of primary node operations within the replica set. Such a configuration insures high availability of the data replicated throughout the replica set.

In one embodiment, a database system can be configured to permit read operations from any node in response to requests from clients. For reads, scalability becomes a function of adding nodes (e.g. servers) and database instances. Within the set of nodes, at least one node is configured as a primary server. A primary server/node provides the system with a writable copy of the database. In one implementation, only a primary node is configured to permit write operations to its database in response to client requests. The primary node processes write requests against its database and replicates the operation/transaction asynchronously throughout the system to connected secondary nodes.

In another example, the group of nodes, primary and secondary nodes operate in conjunction to process and replicate database operations. This group of nodes can be thought of a logical unit, a replica set, for handling database operations. Shown, for example, in FIG. **2** are the basic elements of a replica set, a primary or master node **202** and secondary or slave nodes **208-210**. The primary node's responsibility can transition between nodes **202**, **208**, and **210** within the replica set, permitting operation even in light of failures within the replica set. The secondary nodes **208-210** host replicas of the primary database and are configured to take on the primary role automatically in the event of a failure.

In another example, the primary node receives and performs client writes operations and generates an operation log. Each logged operation is replayed by the secondary nodes bringing the replicated databases into synchronization. In some embodiments, the secondary nodes query the primary node to identify operations that need to be replicated. The replica set and/or individual nodes can be configured to response to read request from clients by directing read request to slave nodes **208-210**.

Clients, for example **204-206**, from the perspective of a distributed database can include any entity requesting database services. A client can include an end-user system requesting database access and/or a connection to the database. An end-user system can request database services through an intermediary, for example an application protocol interface (API). The client can include the API and/or its associated drivers. Additionally, web based services can interact with a distributed database, and the web based services can be a client for the distributed database.

By implementing each shard as a replica set, the shard cluster can provide for high availability and high consistency in the underlying data. In one example, a replica set can be a set of n servers, frequently three or more, each of which contains a replica of the entire data set for the given shard. One of the n servers in a replica set will always be a primary node. If the primary node replica fails, the remaining replicas are configured to automatically elect a new primary node. Each illustrated server can be implemented as a replica set, for example, as discussed in co-pending application Ser. No. 12/977,563 entitled "METHOD AND APPARATUS FOR MAINTAINING REPLICA SETS" filed on Dec. 23, 2010, incorporated herein by reference in its entirety. Other replication methodologies can be used to insure each shard remains available to respond to database requests. In some examples, other multi-node systems can be used to provide redundancy within a sharded database. In one example, master/slave configurations can be employed. In others, various distributed architectures can be used for each shard within the shard cluster. In some embodiments, each replica set can also execute an aggregation engine for receiving aggregation operations. Further, the aggregation engine can further optimize operations within an aggregation operation locally. In some embodiments, an aggregation engine associated with a routing process can identify the potential for local optimizations, and pass the aggregation operation to another aggregation engine being executed locally on a replica set hosting data needed to complete the aggregation operation. Further dependency analysis can be executed locally as well as re-ordering of execution of operations within the aggregation operation.

Shown in FIG. **4** is one example process flow **400** for optimizing an aggregation operation according to one embodiment. In one example, aggregation operations can be received from end-users wishing to access data available on the distributed database. In another example, aggregation operations can be identified by routing process/entity in the database (e.g., **116-118**, FIG. **1**) and communicated by the routing process/entity for execution.

Process **400** begins at **402** with the identification of an aggregation operation. Once an aggregation operation is identified, the operations with the aggregation operation are analyzed at **404** to determine if any optimization can be generated. In some embodiments, aggregation operations are submitted to the database in a standard format. In one example, an aggregation operation is defined by an end-user using an aggregation wrapper. One format includes designation of a database or collection followed by a keyword,

tag, or flag that an aggregate operation is being requested. The operations to be performed are then defined as a group. In some embodiments, the end-users specify the group of operations with a particular order to achieve a specific result. In response to analysis at **404**, which can include, for example, dependency graphing of the data (e.g., data documents, data fields, etc) being accessed by the operations, a re-ordered execution can be defined that limits the amount or data requested and/or accessed. In some embodiments, optimized aggregation operations are generated at **406**. For example, optimized operations can be generated by re-ordering execution. In other examples, specific operations can be identified and elevated or demoted in execution order to reduce the computation burden of executing the aggregation operation.

Once an optimized aggregation operation is generated at **406**, the optimized operations can be communicated at **408** through a distributed database to the systems hosting the data needed for responding the aggregation operation. For example, the operation can be communicated to a plurality of shards in a sharded database. In some embodiments, process **400** can also be executed on the systems hosting the data to provide for further optimizations.

FIG. **5** illustrates an example process **500** for optimizing execution of an aggregation operation, according to one embodiment. In one embodiment, process **500** can be executed by a system (e.g., **700** FIG. **7**) or by an aggregation engine executing on such a system. In some embodiments process **500** can be executed as an independent process, and in others can be executed as part of or in conjunction with other processes (e.g., **400**). Process **500** begins at **502** with analysis of operations within an aggregation operation. The dependencies for the operations in the aggregation are identified and analyzed to determine which data documents and/or data fields need to be carried through the execution in order to generate an output for the aggregation operation. In one example, a dependency graph is generated at **504**. The dependency graph identifies which operations in the aggregation need what data, and further what data is required by subsequent operations. Based on the analysis of the operations for data dependencies, data access requests for any data unnecessary to the execution can be filtered from the aggregation operation at **506**. In some examples, filtering can include generating exclusion operations or commands within the aggregation operation. In other examples, filtering at **506** can include generating exclusion operations and/or commands within one or more operations contained in the aggregation. In other embodiments, the data being returned or forwarded for processing by a subsequent stage can be filtered to only carry forward fields that are actually required for the output.

In some other examples, the dependency analysis at **504** identifies data field and/or document that can be eliminated from an execution. At **506**, data fields and/or document can be culled from execution of the aggregation operation. For example, an unmodified data request may include a plurality of database instances in order to response to the unmodified request. Once certain data fields are eliminated, a system or engine executing process **500** can limit the number of the plurality of database instances that need to receive an optimized version of the operations.

In other examples, returned responses from the plurality of database instances can filter any excluded data, limiting the need to communicate data unnecessary for an output to the requested aggregation operation. In further examples, the evaluation of dependencies and filtering of unnecessary data can take place in conjunction with routing the data request to shards that host the data needed to respond. As discussed, the dependency analysis can be configured to identify shards that are no longer necessary to respond based on the dependency analysis and any routing of data requests can be configured appropriately.

In some other embodiments, filtering can occur when results are being accumulated for projection to another system, for example, as a response. In some further embodiments, additional filtration of data can also occur prior to and/or in conjunction with projection of a data output.

Shown in FIG. **6** is another example process **600** for optimizing execution of an aggregation operation, according to one embodiment. In one embodiment, process **600** can be executed by a system (e.g., **700**, FIG. **7**) or by an aggregation engine executing on such a system. In some embodiments, process **600** can be executed as an independent process, and in others can be executed as part of or in conjunction with other processes (e.g., **400**, **500**). In some implementations process **600** can access data generated from other processes including for example, dependency analysis generated by process **500** at **504**.

Process **600** begins at **602** with analysis of operations within an aggregation operation. In some examples, the aggregation operation includes a set of operations to be performed as listed in an aggregation wrapper. In some examples, the aggregation wrapper and/or operation can be indicated by a keyword, a flag, etc. The operations within the aggregation can be evaluated to identify classes of the operation within the set. In particular, any filtering operations are identified with the set at **604**. Filtering operations identified at **604** include, for example, any operation(s) that when executed reduces the set of data being analyzed, accessed, or transformed during the execution of the aggregation operation. Match$ is one example of a standard operation for filtering data accesses and/or results. Match$ enables definition of conditions on which data is valid to return in response to an operation. In relational database settings, the SQL command identifying "Where [Condition]" can provide similar functionality, however lacking the table structure such SQL operations cannot be executed effectively in a database structured on documents. If filtering operations are identified at **604**, then evaluations on the set of operations can determine if re-ordering of the execution of the set of operations is permitted. In addition or alternatively, evaluation of the set of operations can determine whether re-ordering will reduced the complexity of execution, and/or reduce the volume of data accessed or communicated, for example, during execution of the aggregation operation.

In one example, by re-ordering the operations contained within an aggregation wrapper filtering operations can be executed first or at an earlier position within an aggregation pipeline. Earlier execution of filtering operations can be configured to reduce the amount of data accessed, and potentially the number of distributed systems touched to execute the aggregation operation. In some executions, re-ordering can also be performed at each of the distributed systems that contain data associated with an aggregation operation.

In some embodiments, the re-ordering evaluation can be rule based. In one example, the system executes a rule which causes the system to evaluate the set of operations for conditions that indicate re-ordering can be made and/or will reduce computational burden. For example, a rule can be executed by the aggregation engine to identify filtering operations that appear after data manipulation operations (e.g., sort (configured to re-arrange presentation of data)),

such filtering operations can be executed prior to the data manipulation, for example, to reduce computational burden and/or the amount of data input into the data manipulations. In one embodiment, the rule can be configured to identify $match operations which appear after a sort operation as an operation that can be moved.

In another embodiment, a rule can be configured to identify filtering operations the follow other data manipulation operators. For example, the rule can identify $match following a $project operation. In some embodiments, $project is configured to reshape a data (e.g., document) stream by renaming, adding, or removing fields. The system can executed a $project command to create computed values or sub-objects. Depending on context provided for execution of the project operation, the system can be configured to execute a received $project operation to operate on documents output from a preceding operation as input to the $project operation as part of an aggregation operation. According to some embodiments, context provided with the $project command causes the system executed operations to perform any one or more of the following: modify the input data to include fields from an original input document, exclude fields from the original document, insert computed fields, rename fields, create and populate fields that hold sub-documents, among other options. In some implementations, the system can execute a $project operation to quickly select the fields to include or exclude from a data stream that can make up, for example, a response to a data access request.

In some embodiments, the rule can be configured to identify what computations the $project operations performs on any data stream. For example, as long as the $project does not compute the $match fields, a following $match operation can be executed prior to the $project. In some embodiments, rule execution can also include dependency analysis of the operations within the aggregation. Dependency analysis can enable further re-orderings of operations that eliminate unnecessary data that is not dependent on by subsequent operations. For example, filtering operations typically cannot be re-ordered when appearing before or after group operations, however, if the dependency analysis indicates that the filter only impacts data fields not relied on or used in subsequent group operations, re-ordering can be done. An aggregation engine can be configured to generate a mapping of document and/or data dependencies of the operations within the aggregation pipeline into a dependency graph. In some executions, analysis of the dependency graph enables additional re-orderings.

Once re-orderings has been identified **606** YES, a reorder execution pipeline can be generated at **608**. The reorder execution pipeline can also be evaluated and/or re-evaluated to determine if the reorder pipeline can be further modified. Further if no re-ordering is appropriate **606** NO, the operations within the aggregation pipeline can still be evaluated to identify operations that can be executed together. For example, an aggregation pipeline can be evaluated to identify expansion operations that are configured to expand data being analyzed. In one embodiment, an $unwind operation is configured to flatten BSON documents input into the operation. BSON documents can include arrays stored as fields in the document, and further can contain references to other BSON documents. $unwind flattens the arrays and any references to other documents potentially creating large volumes of data for subsequent operations. Under conventional execution the expansion operation flattens all references and arrays within the input documents and then any subsequent operation specified in the pipeline are performed. Combining execution of the expansion operation with subsequent operators can improve execution. For example, combining an $unwind operation with a subsequent filter operation can improve computational efficiency dramatically.

Having identified combination operations at **610** YES, the execution of operations within the aggregation can be modified at **612**. Once any re-ordering has been identified (e.g., **606** YES), any combinations of operations has been identified (e.g., **610** YES) and the set of operations within the aggregation operation has been modified (e.g., **608** and **612**), the resulting aggregation operation can be communicated to the various systems hosting data for completing its execution at **614**. In some instances re-ordering is not appropriate (e.g., **606** NO), and combination is not warranted (e.g., **610** NO). In such executions, the original aggregation operation can be communicated at **614** to the systems hosting data for its execution.

Process **600** can also be re-executed by each of the systems hosting communicated to at **614**. Local optimizations can be identified by aggregation engines having access to the underlying data referenced by the aggregation operation and/or the individual operations within the aggregation pipeline.

In some embodiments, optimization of aggregation operations can trigger a change in how the system processes specific operations. For example, the system can be configured to execute specific behavior for defined operations when operating on data requests sent outside of an aggregation operation. When identified as executing within an aggregation operation, the system can be configured to implement different behavior for the execution of the operation. In some embodiments, the behavior executed can be dependent upon the ordering of operations and/or the specific operations contained within the aggregation operation.

In one embodiment, the system can be configured to identify $sort operations that appear in conjunction with $limit operations. In response to identification of a $sort operation followed by $ limit, the system can be configured to change the execution of $sort. In one embodiment, the execution of $sort is changed to provide for storage optimizations. In particular, $sort can be executed by the system to organize documents or data results into a heap. The system can be configured to constrain the instantiation of the heap to store result based on the parameters associated with the $limit operation. As discussed in greater detail below, execution of $limit by the system includes specification of a limit parameter that identifies a restriction on a number of data records and/or documents that pass through the aggregation pipeline. When the system identifies an aggregation operation including sort followed by a limit, instead of instantiating a sort tree into which all results would be inserted, the system can instead instantiate a heap and constrain the heap size to the limit specified in the limit operation. According to one embodiment, the constraint of the heap to the size of the limit reduces memory consumption on the node or system which is executing the sort operation.

During execution the system can be configured to fill the elements of the heap, evicting data from the heap as the data is processed and whenever the limit restriction would be exceeded. Once the stream of data input into the sort operation is processed the resulting sorted heap can be emitted either as a result or for further processing by subsequent operations in an aggregation operation. In some further embodiments, the system can be configured to combine the execution of limit and sort. In other embodiments,

the system is configured to identify the combination of limit and sort to modify the default behavior of the sort operation. In one embodiment, the system is configured to execute default behavior for a $sort operation on data by collecting all data responsive to the request at one node (e.g., a routing server/node) from any shards or systems hosting data responsive to the request. The node responsible for processing the data can be configured to instantiate a single data structure and insert any received data into the data structure. In one example, the node instantiates a sort tree and inserts received data into the tree. The default operation can be optimized, for example, by an aggregation engine when analysis executed by the aggregation engine identifies $sort operation being executed within an aggregation wrapper. According to one embodiment, under aggregation optimization, the sort operator can be executed by the system and/or aggregation engine to perform a merge sort operation, rather than the default insertion into a sort tree. For example, a node responsible for aggregating the data returned from a plurality of shards can be configured to execute a multi-way merge on the received data.

In one embodiment, the system and/or an aggregation engine can be configured to analyze the results to automatically identify a vector integrator to produce a multi-way merge of the results. In some embodiments, the vector integrator is a common data element within a set of results. The results can be analyzed to identify common data fields and further to score or rank common data filed based on their presence within the data results. In one embodiment, once the system or aggregation engine identifies the vector on which to merge, that field is then selected out of each line of data, and the system is configured to merge those lines of data into a single sorted data structure.

In one example, the vector integrator can be identified as a top element (e.g., most common or other ranking methodology) of the set of results. In a transaction database, for example, a customer field may be identified as a top-element and be used by the system and/or an aggregation engine executing on the system to merge sort a returned set of results. In some embodiments, multiple vector integrators can be used to merge sort a received set of results. Once a first set of results have been merged into a sorted data structure based on a first vector integrator, any remaining results can be merged into the data structure based on a second vector integrator. A sorted set of results can thus be built from merges made on a set of results by the system or an aggregation engine.

In some implementations, any one of a routing process, data management process, and aggregation engine executing on nodes within the database can be configured to identify and analyze aggregation operation for optimizations. In some embodiments, the processes can be configured to identify optimizations and flag them for optimized execution on other nodes within the database. In other embodiments, routing, management, and aggregation processes can be executed in conjunction, and use, for example, inter-process communication to identify and optimize execution of aggregation operations.

Shown in FIG. **8** is an example process **800** for analyzing operations within an aggregation operation. Process **800** begins at **802** with analysis of a plurality of operations contained within an aggregation operation. In some examples, the aggregation operations can be formatted within an aggregation wrapper as discussed above. The listing of each operation within the aggregation wrapper can also be used to specify an ordering of the execution. Process **800** includes a determination of the execution environment at **804**. For example, the determination can include identifying a sharded database architecture, a replica set implementation, combinations of sharded architecture supported by replica sets, or other data architectures for a distributed database. In one embodiment, an aggregation engine can be configured to perform process **800**. In other embodiments, a routing process and/or data management process can be configured to execute process **800**.

Based on the execution environment identified at **804** operations with the aggregation operations can be identified for optimized execution at **806**. In one embodiment, a sharded environment can be detected at **804** and, for example, a sort operation can be identified at **806**. The sort operations can include a default execution, which when executed by the system can include inserting data (e.g., documents) into a sort tree. Upon identification of the operations at **806**, the node executing the operation can be configured to modify its operation according to the environment in which the operation is being executed at **808**.

In one embodiment, a system can be configured to identify the sort operation within an aggregation operation (e.g., **806**). Once the operation is identified, process **800** continues at **806** by changing the operation of sort from insertion into a sort tree to a merge sort operation, where data results (e.g., documents) from a plurality of nodes hosting data can be more efficiently sorted. As discussed above, execution of sort within an aggregation by the system can be modified to employ vector identifiers to merge sort results rather than perform an insertion sort operation. Thus, the overall execution of the aggregation operation can be improved by reducing the resources required on the system or node responsible for aggregating a data result received from a plurality of nodes hosting the referenced data. In one embodiment, a node on a distributed database system executing a data routing process can also be configured to accept responses from the nodes hosting any requested data. The node executing the routing process can also include an aggregation engine configured to manage optimized execution of, for example, a sort operation within an aggregation operation.

According to other embodiments, additional optimizations can be executed by the system (e.g., an aggregation engine executing on the system). In one embodiment, execution of the $group operation can be optimized based on identification of a preceding $sort operation found within an aggregation pipeline. For example, analysis executed by the aggregation engine on an aggregation operation, prior to or even during execution of the operation within the aggregation operation can identify that a data stream input into a $group operation is a sorted sequence of data. In one example, the aggregation engine or another process can be configured to identify a $sort operation on a set of data where the data is subsequently input into a $group operation. In other examples, the aggregation engine can identify other sorted data sequences based on the operations executed prior to the $group operation.

Execution of $group under default conditions can proceed by the system instantiating buckets for each group of data output by the $group operation. Default operation can be memory intensive as each of the group buckets is maintained during execution to insure a position for all the data elements from an input data stream in a valid output location. When the aggregation engine or other data management process executing on the system determined that the data input into the group operation is sorted, the aggregation and/or management process can be configured to further determine if the incoming data stream (e.g. documents) are sorted by a

key used in establishing the groups for the $group operations. If the data stream is ordered on a group operator identifier (e.g., a group key) then execution of the group operation by the system can limit the resources to hold the output to one group bucket (e.g., data structure) at a time. Once the bucket is filled the bucket can be emitted and a next bucket instantiated to hold data. A one data structure at a time approach enables the system to conserve active memory resources, and can improve the system's ability to handle concurrent requests accordingly.

In one example, a database of documents containing information on rental cars can be accessed by a data request received from a client. If the data request includes an aggregation operation, an aggregation engine can analyze the operation to identify potential optimizations. In particular, a group operation executed within an aggregation can have documents describing cars sorted by month as an input. Where the group identifier is month, the data results will be organized into buckets (i.e., data structures) based on month. Each bucket will be filled in order because the input is sorted on month, thus, the aggregation engine can optimize memory and system resource allocations by only instantiating one bucket at a time.

According to another aspect, execution of an aggregation pipeline can be further optimized to reduce system resources consumed during execution. According to one embodiment, aggregation operations can touch large amounts of data across a plurality of database instances. The database system can be configured to communicate not only the underlying data between nodes, but also pass the returned results through the various subsequent operations specified in the aggregation pipeline. In one embodiment, the database system can be configured to employ hashtables of named fields within the data results, reducing the volume of data operated on, communicated between nodes and/or database instances, and/or reducing the volume of data communicated between stages of execution of the aggregation pipeline. In one example, a hash value can be generated to represent strings (i.e., groups of characters, which can be words, sentences, etc.) within data sets. A hash function can be executed by the system to transform the names of the data fields into hash keys that can be used to look up the actual value for the names of the data fields. As the actual values for the names of the data fields can be irrelevant to generating and returning data in response to a data request, hash values reduces the amount of data being processed. In some examples, the field names being accessed, parsed, and communicated require more resources than the underlying data that, for example, an end user is requesting. In various embodiments, actual field names can be stored once and hash values transmitted and/or communicated to other systems and processes for further processing.

In some embodiments, further optimizations are achievable based on analysis of an aggregation operation and the underlying data upon which the aggregation operation is determined to access. In particular, where the data requested by the aggregation operation or a portion of the aggregation operation targets key index data, an index only stage can be executed by the aggregation engine. The index only execution enables the aggregation engine to process and return results directly from indexed data enabling faster execution. In some examples, indexes may be used by a routing process to access and return results, rather than requiring a request to a database instance hosting the data associated with the indexed record. In some embodiments, any one or more of a routing process, aggregation engine, and data management process can be configured, when executing, to parse an aggregation operation to identify data and/or data fields that the operations within the aggregation access. In one example, the analysis of the aggregation operation includes a single scan through the operations and the criteria specified by the operations.

In conjunction with, for example, data from a configuration server which hosts the metadata describing what data is stored on what nodes in the distributed database, various embodiments of the system can determine what data is being requested by any operation, group of operations, pipeline phase, or even portion of an operation within the aggregation operation. In one example, the system can be configured to analyze aggregation operations against composite index keys, where the composite index keys are made up of multiple data fields concatenated together. Based on analysis of, for example, configuration metadata describing the data available in the database, the system can determine what data fields and/or operations target index data.

In some embodiments, the database system can also include user interfaces that are configured to provide visual indications of potential optimizations to end-users requesting data. For example, as an end-user constructs a data query, the user interface can be configured to evaluate the input query. In another example, the interface can communicate the query to the database system as it is entered in the user interface. These options, among others, enable the system to evaluate the data request to determine if any optimizations can be made. In one environment, the data request is evaluated to identify data requests that operate on indexed data fields. Based on the analysis visual feedback can be provided to, for example, an end-user inputting the data request. The visual feedback can include highlighting and alteration of the font for a given data field displayed within a user interface, to show that the particular data field is an indexed data field. In one embodiment, the aggregation engine can be configured to provide visual feedback and further to optimize execution of the aggregation operation or a portion of the aggregation operation that is targeted to the indexed data. In another embodiment, index only portions of an aggregation operation can be separated into its own phase of execution.

According to other embodiments, non environment aware execution of cumulative operators (e.g., group, sort, etc.) can be memory intensive. In one embodiment, the aggregation engine can be configured to recycle resources used by specific operations during execution of the aggregation pipeline. For example, if the aggregation pipeline includes a group operation, the system can be configured to instantiate a hashmap of the various groups generated as the output of the group operation. In another example, the entire set of results can be required to be instantiated in memory in order to complete execution of the group operation. By recycling those resources as the grouped data is emitted, for example, to another stage in the pipeline or emitted to an end user as data results, the system can optimize execution of the aggregation pipeline. In further examples, sort operations can occupy similar resources during execution. In one example, sort execution on the system can be optimized by removing sorted entries as they are emitted, either to another stage or as end results. In another example, reference-counting pointers used by the system can also be freed as any associated data is emitted during execution of an aggregation pipeline and/or phases of an aggregation pipeline.

According to other embodiments, an aggregation pipeline execution can be viewed from a system execution perspective as a document to document transformation. A data request can capture data in the form of documents, building

a new document from the underlying data to be emitted to a requestor. The execution of the aggregation pipeline can thus include a variety of filter operations, mathematical operations, organizations, etc., generate and project the new document as a result or as an input to a next stage of the aggregation pipeline. In particular, aggregating operations (e.g., group and sort) can be used to define stages of execution for an aggregation pipeline, where the preceding operations up to an aggregating operation can be assigned to a phase and at least the aggregating operation and potentially subsequent operations can be assigned to a subsequent phase. In one embodiment, by separating the operations up to an aggregating operation, the system can manage parallelization of the execution of the phases, by distributing execution of the phases.

In one example, a database manager can be executing on a node within the distributed database hosting a partition of database data. Multiple data threads can be instantiated by the node to handle large input data streams being operated on within an aggregation operation. A input data stream received by an operation within the aggregation can result in, for example, 100 thousand output documents. An aggregation engine executing on the node can be configured to carve the output into 100 subsets of 1000 documents, and execute each subset against its own data thread. The 100 subsets and the respective thread can be executed within the one database instance on the node in parallel until an aggregating operation (e.g., group and sort) processes the result as its input. In some embodiments, the aggregation engine can be configured to scan the aggregation operation and determining independently the extents (i.e. groupings of responsive data values and/or the documents containing them) of the data requested by each operation. In one embodiment, the aggregation engine determines the extents of the operations by instantiating a partition of the B-tree of the data required. In some embodiments, multi-threaded execution can be implemented by the system to scan various partitions of the B-tree generated from the operations and data specified by the aggregation pipeline.

According to another embodiment, an aggregation engine can also be configured to support nested pipeline operations within an aggregation operation. Similar to the nested properties of BSON documents, in which documents can contain reference to other documents or further stored arrays of values within one document, nested pipeline execution can include specification of a sub-ordinate aggregation operation within a first aggregation operation. In one embodiment, referenced data can include arrays and nested arrays within documents stored in the database. A nested pipeline can be configured to target the nested data within the reference documents, for example, by unwinding the array data, generating data matches against that data, and streaming the results to subsequent aggregation operations. By executing a nested pipeline, an aggregation engine can unwind the references array data, generate matches, and generate a new document containing the subset of data, where the new document is passed to subsequent operations in the pipeline. In some embodiments, sub-ordinate pipeline operations can include expansion operators, filtering operators which can be used to collapse or reduce the set of data that needs to be passed to further operations. In another example, a nested pipeline can be used to unwind array data within an aggregation operation where the nested array combines an unwind operation and expression operators that generate a value from the unwound data. In one example, only the values generated are passed to subsequent operations, thus executing a nested pipeline reduces the data passed from one operation to the next, in this example, by unwinding arrayed data, generating a result from the unwound data, and projecting/generating a new document holding the results for subsequent operations.

In other embodiments, execution of aggregation pipeline can be configured to enable complex query behavior without overly complex query operations, for example, like map-reduce. In one embodiment, the aggregation framework can support moving of data using aggregation operations. For example, a data request in an aggregation operation can specify skip value for a condition associated with a skip operation. As discussed below, a skip operation can be configured to ignore a specified number of data elements (e.g., documents) and to pass on subsequent data after that number to the next operation. Used in conjunction with a limit operation (which specifies a number of data element or documents that will be operated on before ignoring any subsequent data) an aggregation operation can specify a data window on which other operations specified in the aggregation operation will operate. A projection operation can generate an output for the specified window. Moving windows of execution on such data streams can be implemented, for example, by an aggregation engine. In one example, the skip condition can be specified by an incrementing variable, and in others the aggregation operation or the moving window portion of the aggregation operation can be recursive until reaching the end of a particular data stream. Moving windows of execution on data streams can be configured to permit data capture and analysis on portions of a data stream, for example, allowing averages, counts, and/or smoothing of data values within each moving window.

In some embodiments, a node or node executing a windowing aggregation can be configured to store state information regarding the execution of the moving window. The state information can be configured to track a place in a respective data stream being processed by the window. In one embodiment, the aggregation engine can be configured to access the state information to enable recursive executions to be expedited. Further, in some implementations "windowing" executions can include caching of a data stream by the aggregation engine for a next recursive execution. Cached data streams and state information can be accessed by, for example, the aggregation engine to speed processing by looking up a place in the data stream where the operation left off.

In some further implementations, an aggregation engine can be configured to enable branched pipeline execution of any operations specified within the aggregation operation. For example, analysis of a pipeline by the system can identify where an operations or phase of a pipeline is configured to emit results to a second collection. In this example, the pipeline (prior to the emit to the second collection) can be duplicated at multiple nodes for execution against the data at that particular node. As each node completes the output, the result can be communicated to the node responsible for the second collection. In another example, branched execution can be executed when the output is being delivered to a permanent collection, that is stored permanently for subsequent access. In such a scenario, pipeline branches can be duplicated across the nodes hosting data referenced by the aggregation operation.

In other embodiments, the analysis engine can identify phases of an aggregation operation based on the data collections needed to response. If the analysis of the aggregation determines that portions of the operations within the aggregation operate on collections independently, the phases

can also be executed in branches. In one example, duplicates of each phase are executed at a variety of nodes to generate results that can be directed to new collections, or in other examples, directed to an end-user requesting the data.

Example Aggregation Framework Implementations

According to one embodiment, the aggregation framework provides a means to calculate aggregate values without having to use complex syntax or computationally intensive operations like map-reduce. While map-reduce is powerful, using map-reduce is more difficult than necessary for many simple aggregation tasks, such as totaling or averaging field values.

The aggregation framework can provide similar functionality to GROUP BY and related SQL operators as well as simple forms of "self joins." Additionally, in some embodiments, the aggregation framework provides projection capabilities to reshape the returned data. For example, using projections and aggregation, the system is configured to add computed fields, create new virtual sub-objects, and extract sub-fields into the top-level of results.

According to various aspects, documents stored in the database can be organized into collections. These documents pass through an aggregation pipeline, which transforms these objects they pass through. Conceptually, various embodiments provide functionality similar to the pipe (i.e. |) operator from UNIX environments which, in one example, was implemented in UNIX to string text filters together. In a shell environment the pipe redirects a stream of characters from the output of one process to the input of the next. According to various aspects, the aggregation framework implements an aggregation pipeline for execution of a series of operations. The aggregation pipeline streams data (e.g., documents) from one pipeline operator to the next to process the data.

In some embodiments, the pipeline operators process a stream of documents and the pipeline behaves as if the operation scans a collection and passes all matching documents into the "top" of the pipeline. In one example, each operator in the pipeline transforms each document as it passes through the pipeline. In some examples, pipeline operators need not produce one output document for every input document: operators may also generate new documents or filter out documents.

In some embodiments, specific operators are defined for use in the aggregation framework. In some examples the operators include any one of more of $project, $match, $limit, $skip, $unwind, $group, and $sort discussed in greater detail below. According to other embodiments, the aggregation framework can also define expression configured to calculate values from documents as they pass through the pipeline and collect these results with calculated values from the other documents that have flowed through the pipeline. In some embodiments, the aggregation framework simplifies the definition of expressions that can be executed on the system. In one example, expressions can be defined in a JSON format using prefixes to establish operation parameters.

Often, expressions are stateless and are only evaluated when executed by, for example, an aggregation engine. Stateless expressions perform operations such as adding the values of two fields together or extracting the year from a date. In other embodiments, expression can retain state information. For example, accumulator expressions retain state during execution. In one example, the $group operator maintains that state (e.g. totals, maximums, minimums, and related data.) as execution of the operation in the pipeline proceeds.

In some embodiments, aggregation operations are defined by an end-user with the aggregate wrapper in a programming shell. In other embodiments, aggregation operations can be defined using an "aggregate" database command. In some examples, aggregate operations are execution on a collection object (i.e., a named group of database documents), which will determine the documents that contribute to the beginning of the aggregation pipeline. The arguments to the aggregate function can specify a sequence of pipeline operators, where each pipeline operator may have a number of operands.

An example aggregation operation is illustrated, including an example data environment of the following format:

```
{
title : "this is my title" ,
author : "bob" ,
posted : new Date( ) ,
pageViews : 5 ,
tags : [ "fun" , "good" , "fun" ] ,
comments :
    [
    { author :"joe" , text : "this is cool" } ,
    { author :"sam" , text : "this is bad" }
other : { foo : 5 }
}
```

The example aggregation operation (below) pivots data to create a set of author names grouped by tags applied to an article:

```
db.article.aggregate(
{ $project : {
    author : 1,
    tags : 1,
} },
{ $unwind : "$tags" },
{ $group : {
_id : { tags : 1 },
authors : { $addToSet : "$author" }
}}
);
```

The example operation employs an aggregate wrapper around the database command aggregate. The aggregation pipeline begins execution with the collection article and selects the author and tags fields using the $project aggregation operator, and runs the $unwind and $group on these fields to pivot the data. The result of the aggregation operation returns a document with two fields: result which holds an array of documents returned by the pipeline ok which holds the value 1, indicating success, or another value if there was an error. In some embodiments, the result is returned a new document, which can be stored in the database.

According to some embodiments, the aggregation engine can be configured to partition the execution of an aggregation pipeline into multiple parts. For example, when an aggregation operation is executed by the system against a sharded collection, the aggregation pipeline can be split into multiple parts. For example, the aggregation engine can be configured separate the aggregation pipeline into phases based on the operations being executed. In one example, the aggregation engine is configured to identify the first instance of a $group or $sort operation to a first shard and assigned all preceding operations including the $group or $sort to a first phase. In another example, the remaining operations can be executed as a second phase. In one embodiment, the

aggregation engine can be configured to assign the phases of the aggregation pipeline to different shards, increasing the parallelism of the execution of the aggregation operations.

Further, the aggregation engine can be configured to execute the second part of the pipeline on the routing server, or can be configured to pass the second part of the pipeline to the routing process. The second phase of the pipeline can be configured to also include the first $group or $sort and any remaining pipeline operators. The second phase of the pipeline can be executed against the results received from the shards executing the first part of the pipeline.

In one example, the aggregation engine connected to the data routing processes is configured to merge results obtained from execution of $sort operations from any of the data shards accessed to respond to the aggregation operation. In another example, the $group operation, brings any "sub-totals" from any of the data shards accessed and then combines them. Additionally, the result of the $group and the various sub-totals can be structures. For example, an $avg expression is configured to maintain a total and count for each shard accessed and the aggregation engine on the routing server can be configured to combine these values and then divides by the aggregate total.

Example Aggregation Framework Operators

As discussed, the aggregation framework provides the ability to project, process, and/or control the output of the query, without using computationally intensive operations or complex syntax, like in map-reduce operations. In some embodiments, aggregation uses a syntax that resembles the same syntax and form as other database queries.

The database system can be configured to make the aggregation framework and aggregation operations accessible by way of a call to a function, e.g., aggregate( ). A variety of examples discussed herein use this function and syntax, the database system and aggregate function identify other syntaxes for an aggregation operation. In one example aggregate( ) can be a wrapper around the database command aggregate. Therefore, the following prototype aggregate command can be interpreted by the system as invoking an aggregation operation:

```
db.people.aggregate( { [pipeline] } )
db.runCommand( { aggregate: "people", { [pipeline] } } )
```

In one embodiment, a system and/or aggregation engine is configured to execute these operations to perform aggregation routines on the collection named "people." "[pipeline]" illustrates a placeholder for the aggregation pipeline definition, which can include a variety of operations, some specific examples are discussed in greater detail below.

According to some embodiments, an aggregation pipeline can be defined to operate on a named collection of documents within a distributed database. In some examples, pipeline operators are listed in an array for execution by the system. Conceptually, data (e.g., documents) pass through these operators in a sequence. Examples of the operators are discussed in greater detail below.

One example operator is $project. $project can be configured to reshape a data or document stream by renaming, adding, or removing fields. $project can also be executed to create computed values or sub-objects. $project can be configured to: include fields from the original document, exclude fields from the original document, insert computed fields, rename fields, create and populate fields that hold sub-documents. Various examples include the use of $project to quickly select the fields that a user wants to have the system include or exclude from a generated response or output during an execution of an aggregation operation. An example operation:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        author : 1 ,
    }}
);
```

includes the title field and the author field in a document that is returned from the execution of the aggregation pipeline. In this example, because the first field specification is an inclusion, $project is in "inclusive" mode. In inclusive mode $project can be configured to return only the fields explicitly included (and the _id field.) In some embodiments, the _id field is always included by default in the inclusive mode. However, users can explicitly exclude _id as follows:

```
db.article.aggregate(
    { $project : {
        _id : 0 ,
        title : 1 ,
        author : 1
    }}
);
```

In this example, the projection resulting from execution excludes the _id field but includes the title and author fields. In some embodiments, default execution specifies that a field inclusion in a projection will not create a field that does not exist in a document from the collection. In other embodiments, default execution specifies that in the exclusion mode, the $project returns all fields except the ones that are explicitly excluded.

Another example operation includes:

```
db.article.aggregate(
    { $project : {
        comments : 0 ,
        other : 0
    }}
    );
```

In this example, the projection propagates all fields except for the comments and other fields along the pipeline. In various embodiments, execution of $project is configured to trigger exclusive mode when the first field in the projection (that is not _id) is an exclusion. In other examples, when the first field is an inclusion the projection is executed in inclusive mode. In some embodiments, projections can also add computed fields to the document stream passing through the pipeline. In some examples, a computed field can use any of the expression operators. An example operation includes:

```
db.article.aggregate(
    { $project : {
        title : 1,
        doctoredPageViews : { $add:["$pageViews", 10] }
    }}
);
```

In this example, the field doctoredPageViews represents the value of the pageViews field after adding 10 to the original field using the $add. In some implementations, the

system can be configured to require that the expression that defines the computed field be enclosed in braces, so that it resembles an object and conforms to JavaScript syntax.

In another example operation:

```
db.article.aggregate(
    { $project : {
        title : 1,
        page_views : "$pageViews",
        bar : "$other.foo"
    }}
);
```

When executed the operation renames the pageViews field to page_views, and renames the foo field in the other sub-document as the top-level field bar. In this example, the field references used for renaming fields are direct expressions and do not use an operator or surrounding braces. In various embodiments, aggregation field references can use dotted paths to refer to fields in nested documents.

The following example, when executed by the system, creates a new object-valued field named stats that holds a number of values:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        stats : {
            pv : "$pageViews",
        foo : "$other.foo",
            dpv : { $add:["$pageViews", 10] }
        }
    }}
);
```

This example projection includes the title field and places $project into "inclusive" mode. Then, it creates the stats documents with the following fields: pv which includes and renames the pageViews from the top level of the original documents, foo which includes the value of other.foo from the original documents, dpv which is a computed field that adds 10 to the value of the pageViews field in the original document using the $add aggregation expression.

In some embodiments, projections output fields in the same order that they appeared in the input. Further, in some embodiments, when the aggregation framework adds computed values to a document, they will follow all fields from the original and appear in the order that they appeared in the $project statement.

Another example operation is $match. $mach provides a query-like interface to filter documents out of the aggregation pipeline. The $match drops documents that do not match the condition from the aggregation pipeline, and it passes documents that match along the pipeline unaltered.

In some embodiments, the syntax passed to the $match is identical to the query syntax. Consider the following prototype form:

```
db.article.aggregate(
    { $match : <match-predicate> }
);
```

The following example performs a field equality test:

```
db.article.aggregate(
    { $match : { author : "dave" } }
);
```

The example operation returns documents where the author field holds the value "dave". Consider the following example, which performs a range test:

```
db.article.aggregate(
    { $match : { score : { $gt : 50, $lte : 90 } } }
);
```

in this example, all documents return when the score field holds a value that is greater than 50 and less than or equal to 90.

In another embodiment, $limit is provided. $limit can be configured to restrict the number of documents that pass through the $limit in the pipeline. In one example, $limit takes a single numeric (positive whole number) value as a parameter. Once the specified number of documents pass through the pipeline operator, no more will. In some embodiments, $limit can be recognized by the system as a filtering operation. Consider the following example statement:

```
db.article.aggregate(
    { $limit : 5 }
);
```

this operation returns only the first 5 documents passed to it from by the pipeline. In this example $limit has no effect on the content of the documents it passes.

In another embodiment, a $skip operator is provided. $skip when executed skips over the specified number of documents that pass through the $skip operation in the pipeline before passing all of the remaining input. In one example, $skip takes a single numeric (positive whole number) value as a parameter. Once the operation has skipped the specified number of 25 documents and/or data elements, it passes all the remaining documents/data along the pipeline without alteration. In some embodiments, $skip can be recognized by the system as a filtering operation. Consider the following example statement:

```
db.article.aggregate(
    { $skip : 5 }
);
```

this operation skips the first 5 documents passed to it by the pipeline. In some embodiments, $skip is configured to no effect on the content of the documents it passes along the pipeline upon execution.

In another embodiment a $unwind operator is provided. In one example, $unwind peels off the elements of an array individually, and returns a stream of documents. $unwind when executed can return one document for every member of the unwound array within every source document. Consider the following example statement:

```
db.article.aggregate(
    { $project : {
        author : 1 ,
        title : 1 ,
        tags : 1
    }},
    { $unwind : "$tags" }
);
```

In the above aggregation, $project selects (inclusively) the author, title, and tags fields, as well as the _id field implicitly. Then the pipeline passes the results of the projection to the $unwind operator, which will unwind the tags field. This operation may return a sequence of documents that resemble the following for a collection that contains one document holding a tags field with an array of 3 items. (In some examples, the dollar sign (i.e. $) must proceed the field specification handed to the $unwind operator.)

```
{
    "result" : [
        {
            "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
            "title" : "this is my title",
            "author" : "bob",
            "tags" : "fun"
        },
        {
            "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
            "title" : "this is my title",
            "author" : "bob",
            "tags" : "good"
        },
        {
            "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
            "title" : "this is my title",
            "author" : "bob",
            "tags" : "fun"
        }
    ],
    "OK" : 1
}
```

as shown, a single document becomes 3 documents: each document is identical except for the value of the tags field. Each value of tags is one of the values in the original "tags" array.

The following behaviors are present in some embodiments of $unwind: $unwind can be most useful in combination with $group; the effects of an unwind can be undone with the $group pipeline operators; if a target field is specified for $unwind that does not exist in an input document, the document passes through $unwind unchanged; if a target field is specified for $unwind that is not an array, aggregate( ) generates an error; and if a target field is specified for $unwind that holds an empty array ([ ]), then the document passes through unchanged; and $group groups documents together for the purpose of calculating aggregate values based on a collection of documents. Practically, group often supports tasks such as average page views for each page in a website on a daily basis.

In one example, the output of $group depends on what identifier is provided with the group operator. The system requires that the group operation be entered by specifying an identifier (i.e. a _id field) for the group the system generates within a pipeline. The system can require specification of a single field from the documents in the pipeline, a previously computed value, or an aggregate key made up from several incoming fields. Group expression, can require an _id field. In some examples, the system can require specification of the _id field as a dotted field path reference, a document with multiple fields enclosed in braces (i.e. { and}), or a constant value.

In one embodiment, $project can rename the grouped field after an $group operation. Consider the following example statement:

```
db.article.aggregate(
    { $group : {
        _id : "$author",
        docsPerAuthor : { $sum : 1 },
        viewsPerAuthor : { $sum : "$pageViews" }
    }}
);
```

This example groups by the author field and computes two fields, the first docsPerAuthor is a counter field that adds one for each document with a given author field using the $sum function. The viewsPerAuthor field is the sum of all of the pageViews fields in the documents for each group. Each field defined for the $group can be configured to use one of the group aggregation functions listed below to generate its composite value.

In one embodiment, $addToSet operation is provided. The $addToSet operation can be configured to return an array of all the values found in the selected field among the documents in that group. Each unique value only appears once in the result set. In another embodiment, $first operation is provided. The $first operation can be configured to return the first value it sees for its group. In another embodiment, $last operation is provided. The $last operation can be configured to return the last value it sees for its group. In another embodiment, $max operation is provided. The $max operation can be configured to return the highest value among all values of the field in all documents selected by this group. In another embodiment, $min operation is provided. The $min operation can be configured to return the lowest value among all values of the field in all documents selected by this group. In another embodiment, $avg operation is provided. The $avg operation can be configured to return the average of all values of the field in all documents selected by this group.

In another embodiment, $push operation is provided. The $push operation can be configured to return an array of all the values found in the selected field among the documents in that group. A value may appear more than once in the result set if more than one field in the grouped documents has that value. In another embodiment, $sum operation is provided. The $sum operation can be configured to return the sum of all the values for a specified field in the grouped documents, as in the second use above.

In alternate embodiments, if the operator includes specification of a value as an argument, $sum will increment this field by the specified in same value for every document in the grouping. Typically, as in the first use above, specify a value of 1" in order to count members of the group. In another embodiment, $sort operation is provided. Upon execution, $sort sorts all input documents and returns them to the pipeline in sorted order. Consider the following prototype:

```
db.<collection-name>.aggregate(
    { $sort : { <sort-key> } }
);
```

this example sorts the documents in the collection named <collection-name>, according to the key and specification in the {<sort-key>} document. The sorting configuration can be identical to the specification of an index. To operate within a document, the system accepts specification of a field or fields by and a value of 1 or −1 to specify an ascending or descending sort respectively.

An Example Operation:

```
db.users.aggregate(
    { $sort : { age : -1, posts: 1 } }
);
```

sorts the documents in the users collection, in descending order according by the age field and then in ascending order according to the value in the posts field. In some embodiments, expression operators calculate values within the aggregation framework. According to one embodiment, expression operators include Boolean, Comparison, Arithmetic, String, and Multi-Expression operations.

In one embodiment, Boolean operators are configured to accept Booleans as arguments and return Booleans as results. In some implementations, Boolean operators convert non booleans to Boolean values according to the BSON standards. For example, "Null," undefined, and "zero" values can be "false," while non-zero numeric values, strings, dates, objects, and other types can be "true." In one embodiment, $and is configured to take an array and returns true if all of the values in the array are true. Otherwise $and returns false. In some embodiments, $not is configured to return the Boolean opposite value passed to it. When passed a true value, $not returns false; when passed a false value, $not returns true. In some embodiments, $or not is configured to take an array and returns true if any of the values in the array are true. Otherwise $or returns false.

According to one embodiment, comparison operators perform comparisons between two values and return a Boolean, in most cases, reflecting the result of that comparison. Comparison operators can be configured to take an array with a pair of values. The system can compare numbers, strings, and dates. In one example, except for $cmp, comparison operators return a Boolean value. $cmp can be configured to return an integer. In some embodiments, $cmp is configured to take two values in an array, either a pair of numbers, a pair of strings, or a pair of dates, and returns an integer. In one example, the returned value is: a negative number if the first value is less than the second; a positive number if the first value is greater than the second; and 0 if the two values are equal. In some embodiments, $eq is configured to take two values in an array, either a pair of numbers, a pair of strings, or a pair of dates, and returns an integer. In one example, the returned value is: true when the values are equivalent; and false when the values are not equivalent.

In some embodiments, $gt is configured to take two values in an array, either a pair of numbers, a pair of strings, or a pair of dates, and returns an integer. In one example, the returned value is: true when the first value is greater than the second value; and false when the first value is less than or equal to the second value. In some embodiments, $gte is configured to take two values in an array, either a pair of numbers, a pair of strings, or a pair of dates, and returns an integer. In one example the returned value is: true when the first value is greater than or equal to the second value; and false when the first value is less than the second value. In some embodiments, $lt is configured to take two values in an array, either a pair of numbers, a pair of strings, or a pair of dates, and returns an integer. In one example, the returned value is: true when the first value is less than the second value; and false when the first value is greater than or equal to the second value. In some embodiments, $lte is configured to take two values in an array, either a pair of numbers, a pair of strings, or a pair of dates, and returns an integer. In one example, the returned value is: true when the first value is less than or equal to the second value; and false when the first value is greater than the second value. In some embodiments, $ne is configured to take two values in an array, either a pair of numbers, a pair of strings, or a pair of dates, and returns an integer. In one example the returned value is: true when the values are not equivalent; and false when the values are equivalent.

According to one embodiment, arithmetic operators are configured to perform operations on numbers. In some embodiments, $add is configured to take an array of numbers and adds them together, returning the sum. In one example, if the array contains a string, $add concatenates all items and returns the result as a string. If the array contains a date and no strings, $add treats all numbers as a quantity of days and adds them to the date. The result has the date type. In some embodiments, $divide is configured to take an array that contains a pair of numbers and returns the value of the first number divided by the second number.

In some embodiments, $mod is configured to take an array that contains a pair of numbers and returns the remainder of the first number divided by the second number. In some embodiments, $mod is configured to take an array of numbers and multiples them, returning the resulting product. In some embodiments, $multiply is configured to take an array of numbers and multiples them, returning the resulting product. In some embodiments, $subtract is configured to take an array that contains a pair of numbers and subtracts the second from the first, returning their difference. In some examples where the first entry in the array is a date, $subtract treats the second entry, a number, as a number of days and decrements the date, returning the resulting date.

According to another embodiment, string operators are provided. The string operators can be configured to manipulate strings within projection expressions. In some embodiments, $strcasecmp is configured to take in two strings and return a number, of JavaScript type "long." $strcasecmp is positive if the first string is "greater than" the second and negative if the first string is "less than" the second. $strcasecmp returns 0 if the strings are identical. In some embodiments, $substr is configured to take a string and two numbers. The first number represents the number of characters in the string to skip, and the second number specifies the number of characters to return from the string. In some embodiments, $toLower is configured to take a single string and converts that string to lowercase, returning the result. All uppercase letters become lowercase. In some embodiments, $toUpper is configured to take a single string and converts that string to uppercase, returning the result. All lowercase letters become uppercase. In one embodiment, date operators are provided. Date operators can be configured to take a "Date" typed value as a single argument and return a JavaScript "long" number. In one embodiment, $dayOfMonth operator is provided. $dayOfMonth can be configured to take a date and return the day of the month as a number between 1 and 31. In one embodiment, $dayOfWeek operator is provided. $dayOfWeek can be configured to take a date and return the day of the week as a number between 1 and 7. In one embodiment, $dayOfYear operator is provided. $dayOfYear can be configured to take a date and

return the day of the year as a number between 1 and 366. In one embodiment, $hour operator is provided. $hour can be configured to take a date and return the hour between 0 and 23. In one embodiment, $minute operator is provided. $minute can be configured to take a date and return the minute between 0 and 59. In one embodiment, $month operator is provided. $month can be configured to take a date and return the month as a number between 1 and 12.

In one embodiment, $second operator is provided. $second can be configured to take a date and return the second between 0 and 59. In one embodiment, $week operator is provided. $week can be configured to take a date and return the week of the year as a number between 0 and 53. Weeks start on Sundays and the days before the first Sunday of the year are in "week 0." In one embodiment, $year operator is provided. $year can be configured to take a date and return a four digit number. In one embodiment, $isoDate operator is provided. $isoDate converts a document that contains date constituents into an date-typed object (i.e. in ISODate format.)

In one example $isoDate takes the following form:

```
$isoDate:{$year: <year>,
    $month: <month>,
    $dayOfMonth: <dayOfMonth>,
    $hour: <hour>,
    $minute: <minute>,
$second: <second>
}
```

In another embodiment, multi-expressions are provided. Multi-expressions can be configured to accept multiple expressions and evaluate the inputs to generate a result. For example, $ifNull takes an array with two expressions and returns the first expression if it evaluates to a non-false value. Otherwise, $ifNull returns the second expression's value. In another embodiment, $cond takes an array with three expressions, where the first expression evaluates to a Boolean value. If the first expression evaluates to true, $cond returns the value of the second expression. If the first expression evaluates to false, $cond evaluates and returns the third expression.

Example System Implementations

Various aspects and functions described herein, in accord with aspects of the present invention, may be implemented as hardware, software, or a combination of hardware and software on one or more computer systems. There are many examples of computer systems currently in use. Some examples include, among others, network appliances, personal computers, workstations, mainframes, networked clients, servers, media servers, application servers, database servers, web servers, and virtual servers. Other examples of computer systems may include mobile computing devices, such as cellular phones and personal digital assistants, and network equipment, such as load balancers, routers and switches. Additionally, aspects in accord with the present invention may be located on a single computer system or may be distributed among one or more computer systems connected to one or more communication networks.

For example, various aspects and functions may be distributed among one or more computer systems configured to provide a service to one or more client computers, or to perform an overall task as part of a distributed system. Additionally, aspects may be performed on a client-server or multi-tier system that includes components distributed among one or more server systems that perform various functions. Thus, the invention is not limited to executing on any particular system or group of systems. Further, aspects may be implemented in software, hardware or firmware, or any combination thereof. Thus, aspects in accord with the present invention may be implemented within methods, acts, systems, system placements and components using a variety of hardware and software configurations, and the implementation is not limited to any particular distributed architecture, network, or communication protocol. Furthermore, aspects in accord with the present invention may be implemented as specially programmed hardware and/or software.

FIG. **7** shows a block diagram of a distributed computer system **700**, in which various aspects and functions in accord with the present invention may be practiced. The distributed computer system **700** may include one more computer systems. For example, as illustrated, the distributed computer system **700** includes three computer systems **702**, **704** and **706**. As shown, the computer systems **702**, **704** and **706** are interconnected by, and may exchange data through, a communication network **708**. The network **708** may include any communication network through which computer systems may exchange data. To exchange data via the network **708**, the computer systems **702**, **704**, and **706** and the network **708** may use various methods, protocols and standards including, among others, token ring, Ethernet, Wireless Ethernet, Bluetooth, TCP/IP, UDP, HTTP, FTP, SNMP, SMS, MMS, SS7, JSON, XML, REST, SOAP, CORBA HOP, RMI, DCOM and Web Services.

Computer systems **702**, **704** and **706** may include mobile devices such as cellular telephones. The communication network may further employ one or more mobile access technologies including 2nd (2G), 3rd (3G), 4th (4G or LTE) generation radio access for cellular systems, WLAN, Wireless Router (WR) mesh, and other communication technologies. Access technologies such as 2G, 3G, 4G and LTE and future access networks may enable wide area coverage for mobile devices. For example, the network may enable a radio connection through a radio network access such as Global System for Mobil communication (GSM), General Packet Radio Services (GPRS), Enhanced Data GSM Environment (EDGE), Wideband Code Division Multiple Access (WCDMA), among other communication standards. Network may include any wireless communication mechanism by which information may travel between the devices **704** and other computing devices in the network.

To ensure data transfer is secure, the computer systems **702**, **704** and **706** may transmit data via the network **708** using a variety of security measures including TSL, SSL or VPN, among other security techniques. While the distributed computer system **700** illustrates three networked computer systems, the distributed computer system **700** may include any number of computer systems, networked using any medium and communication protocol.

Various aspects and functions in accord with the present invention may be implemented as specialized hardware or software executing in one or more computer systems including the computer system **702** shown in FIG. **7**. As depicted, the computer system **702** includes a processor **710**, a memory **712**, a bus **714**, an interface **716** and a storage system **718**. The processor **710**, which may include one or more microprocessors or other types of controllers, can perform a series of instructions that manipulate data. The processor **710** may be a well-known, commercially available processor such as an Intel Pentium, Intel Atom, ARM Processor, Motorola PowerPC, SGI MIPS, Sun UltraSPARC, or Hewlett-Packard PA-RISC processor, or may be any other type of processor or controller as many other processors and controllers are available. As shown, the

processor **710** is connected to other system placements, including a memory **712**, by the bus **714**.

The memory **712** may be used for storing programs and data during operation of the computer system **702**. Thus, the memory **712** may be a relatively high performance, volatile, random access memory such as a dynamic random access memory (DRAM) or static memory (SRAM). However, the memory **712** may include any device for storing data, such as a disk drive or other non-volatile storage device, such as flash memory or phase-change memory (PCM). Various embodiments in accord with the present invention can organize the memory **712** into particularized and, in some cases, unique structures to perform the aspects and functions disclosed herein.

Components of the computer system **702** may be coupled by an interconnection element such as the bus **714**. The bus **714** may include one or more physical busses (for example, busses between components that are integrated within a same machine), and may include any communication coupling between system placements including specialized or standard computing bus technologies such as IDE, SCSI, PCI and InfiniBand. Thus, the bus **714** enables communications (for example, data and instructions) to be exchanged between system components of the computer system **702**.

Computer system **702** also includes one or more interfaces **716** such as input devices, output devices and combination input/output devices. The interface devices **716** may receive input, provide output, or both. For example, output devices may render information for external presentation. Input devices may accept information from external sources. Examples of interface devices include, among others, keyboards, mouse devices, trackballs, microphones, touch screens, printing devices, display screens, speakers, network interface cards, etc. The interface devices **716** allow the computer system **702** to exchange information and communicate with external entities, such as users and other systems.

Storage system **718** may include a computer-readable and computer-writeable nonvolatile storage medium in which instructions are stored that define a program to be executed by the processor. The storage system **718** also may include information that is recorded, on or in, the medium, and this information may be processed by the program. More specifically, the information may be stored in one or more data structures specifically configured to conserve storage space or increase data exchange performance. The instructions may be persistently stored as encoded signals, and the instructions may cause a processor to perform any of the functions described herein. A medium that can be used with various embodiments may include, for example, optical disk, magnetic disk or flash memory, among others. In operation, the processor **710** or some other controller may cause data to be read from the nonvolatile recording medium into another memory, such as the memory **712**, that allows for faster access to the information by the processor **710** than does the storage medium included in the storage system **718**. The memory may be located in the storage system **718** or in the memory **712**. The processor **710** may manipulate the data within the memory **712**, and then copy the data to the medium associated with the storage system **718** after processing is completed. A variety of components may manage data movement between the medium and the memory **712**, and the invention is not limited thereto.

Further, the invention is not limited to a particular memory system or storage system. Although the computer system **702** is shown by way of example as one type of computer system upon which various aspects and functions in accord with the present invention may be practiced, aspects of the invention are not limited to being implemented on the computer system, shown in FIG. **6**. Various aspects and functions in accord with the present invention may be practiced on one or more computers having different architectures or components than that shown in FIG. **7**. For instance, the computer system **702** may include specially-programmed, special-purpose hardware, such as for example, an application-specific integrated circuit (ASIC) tailored to perform a particular operation disclosed herein. Another embodiment may perform the same function using several general-purpose computing devices running MAC OS System X with Motorola PowerPC processors and several specialized computing devices running proprietary hardware and operating systems.

The computer system **702** may include an operating system that manages at least a portion of the hardware placements included in computer system **702**. A processor or controller, such as processor **710**, may execute an operating system which may be, among others, a Windows-based operating system (for example, Windows NT, Windows 2000/ME, Windows XP, Windows 7, or Windows Vista) available from the Microsoft Corporation, a MAC OS System X operating system available from Apple Computer, one of many Linux-based operating system distributions (for example, the Enterprise Linux operating system available from Red Hat Inc.), a Solaris operating system available from Sun Microsystems, or a UNIX operating systems available from various sources. Many other operating systems may be used, and embodiments are not limited to any particular operating system.

The processor and operating system together define a computing platform for which application programs in high-level programming languages may be written. These component applications may be executable, intermediate (for example, C# or JAVA bytecode) or interpreted code which communicate over a communication network (for example, the Internet) using a communication protocol (for example, TCP/IP). Similarly, functions in accord with aspects of the present invention may be implemented using an object-oriented programming language, such as SmallTalk, JAVA, C++, Ada, or C# (C-Sharp). Other object-oriented programming languages may also be used. Alternatively, procedural, scripting, or logical programming languages may be used.

Additionally, various functions in accord with aspects of the present invention may be implemented in a non-programmed environment (for example, documents created in HTML, XML or other format that, when viewed in a window of a browser program, render aspects of a graphical-user interface or perform other functions). Further, various embodiments in accord with aspects of the present invention may be implemented as programmed or non-programmed placements, or any combination thereof. For example, a web page may be implemented using HTML while a data object called from within the web page may be written in C++. Thus, the invention is not limited to a specific programming language and any suitable programming language could also be used.

It is to be appreciated that embodiments of the methods and apparatuses discussed herein are not limited in application to the details of construction and the arrangement of components set forth in the following description or illustrated in the accompanying drawings. The methods and apparatuses are capable of implementation in other embodiments and of being practiced or of being carried out in various ways. Examples of specific implementations are provided herein for illustrative purposes only and are not intended to be limiting. In particular, acts, elements and

features discussed in connection with any one or more embodiments are not intended to be excluded from a similar role in any other embodiments.

Also, the phraseology and terminology used herein is for the purpose of description and should not be regarded as limiting. Any references to embodiments or elements or acts of the systems and methods herein referred to in the singular may also embrace embodiments including a plurality of these elements, and any references in plural to any embodiment or element or act herein may also embrace embodiments including only a single element. References in the singular or plural form are not intended to limit the presently disclosed systems or methods, their components, acts, or elements. The use herein of "including," "comprising," "having," "containing," "involving," and variations thereof is meant to encompass the items listed thereafter and equivalents thereof as well as additional items. References to "or" may be construed as inclusive so that any terms described using "or" may indicate any of a single, more than one, and all of the described terms. Any references to front and back, left and right, top and bottom, upper and lower, and vertical and horizontal are intended for convenience of description, not to limit the present systems and methods or their components to any one positional or spatial orientation.

Having thus described several aspects of at least one embodiment of this invention, it is to be appreciated that various alterations, modifications, and improvements will readily occur to those skilled in the art. Such alterations, modifications, and improvements are intended to be part of this disclosure, and are intended to be within the spirit and scope of the invention. Accordingly, the foregoing description and drawings are by way of example only.

What is claimed is:

1. A computer implemented method for executing aggregation operations in a non-relational database architecture, the method comprising the acts of:

receiving, at a database routing system, a plurality of database requests from network entities;

routing, by the database routing system, the plurality of database requests to at least one of a plurality of database access managers over a communication network;

automatically identifying, by an aggregation engine, an aggregation operation within the plurality of database requests, wherein the aggregation operation includes a plurality of data operations, and wherein the act of identifying includes an act of identifying a sequence of execution wherein an execution of a first operation in the sequence of execution enables optimization of a subsequent operation in the sequence of execution;

determining, by the aggregation engine, an optimization to minimize computation time associated with execution of the plurality of data operations; and

modifying, by the aggregation engine, a default execution of at least one of the plurality of data operations to optimize execution of the at least one of the plurality of data operations.

2. The method of claim **1**, wherein the act of identifying the sequence of execution includes identifying a result of the first operation generates an ordered set of results used as input by at least one subsequent operation including a second operation.

3. The method of claim **2**, wherein the second operation includes a group operation, and the act of modifying the default execution of at least one of the plurality of data operations to optimize execution of the at least one of the plurality of data operations includes modifying the execution of the group operation based on analysis of constraints specified by the first operation.

4. The method of claim **3**, wherein the constraints imposed include specification of a key for ordering a data stream.

5. The method of claim **3**, wherein modifying execution of the group operation includes modifying default operation of the group operation to limit instantiation of data structures in executing memory to hold data output of the group operation.

6. The method of claim **1**, wherein the act of identifying includes identifying a sort operation that generates an ordered set of results upon execution.

7. The method of claim **1**, wherein the aggregation operation includes the plurality of data operations and a sequence of execution, and the act of identifying includes identifying a sequence of execution wherein execution of an operation in the sequence permits optimization of a preceding operation.

8. The method of claim **7**, wherein the operation is configured to filter a set of input data to produce a reduced set of output data, and the act of modifying includes modifying a default operation of the preceding operation based on filter criteria specified in the first operation.

9. The method of claim **8**, wherein the operation is a limit operation, and the act of modifying comprises modifying default operation of a sort operation within the plurality of data operations to minimize resources instantiated during execution of the sort operation.

10. The method of claim **8**, further comprising an act of communicating, by the aggregation engine, a modifying version of the aggregation operation having at least one optimization for execution of the plurality of data operations.

11. The method of claim **1**, further comprising an act of determining a data architecture for the data stored on a database.

12. The method of claim **11**, wherein the modifying, by the aggregation engine, the default execution of at least one of the plurality of data operations to optimize execution of the at least one of the plurality of data operations is responsive to the determined data architecture.

13. The method of claim **12**, wherein determining the data architecture includes determining the data architecture includes data shards, and the act of modifying includes modifying the default execution of sort operation to sort by merging data results returned from a plurality of the data shards.

14. A distributed database system for optimizing aggregation operations in a non-relational database architecture, the system comprising:

at least one processor operatively connected to a memory, the processor when executing is configured to execute a plurality of system components, wherein the plurality of system components include:

a router component configured to:

receive database requests from network entities, and

route the plurality of database requests to a plurality of database access managers over a communication network;

an aggregation engine configured to:

automatically identify an aggregation operation within the plurality of database requests, wherein the aggregation operation includes a plurality of data operations,

determine an optimization to minimize computation time associated with execution of the plurality of data operations,

modify a default execution of at least one of the plurality of data operations to optimize execution of at least one of the plurality of data operations, and

communicate the aggregation operation having at least one optimization; and

wherein the plurality of database managers are configured to execute the modified operations and return the resultant data from the distributed database to the network entities.

15. The distributed database system of claim **14**, wherein the aggregation operation includes the plurality of data operations and a sequence of execution, and wherein the aggregation engine is further configured to identify a sequence of execution wherein execution of a first operation in the sequence permits optimization of a subsequent operation.

16. The distributed database system of claim **15**, wherein the aggregation engine is further configured to identify that a result of the first operation generates an ordered set of results used as input by at least one subsequent operation including a second operation.

17. The distributed database system of claim **16**, wherein the second operation includes a group operation, and the aggregation engine is further configured to modify the execution of the group operation based on analysis of constraints specified by the first operation.

18. The distributed database system of claim **17**, wherein the constraints imposed include specification of a key for ordering a data stream.

19. The distributed database system of claim **17**, wherein the aggregation engine is further configured to modify default operation of the group operation to limit instantiation of data structures in executing memory to hold data output of the group operation.

20. The distributed database system of claim **15**, wherein the aggregation engine is further configured to identify a sort operation that generates an ordered set of results upon execution.

21. The distributed database system of claim **14**, wherein the aggregation operation includes the plurality of data operations and a sequence of execution, and the act of identifying includes identifying a sequence of execution wherein execution of an operation in the sequence permits optimization of a preceding operation.

22. The distributed database system of claim **21**, wherein the operation is configured to filter a set of input data to produce a reduced set of output data, and the aggregation engine is further configured to modify the default operation of the preceding operation based on filter criteria specified in the first operation.

23. The distributed database system of claim **22**, wherein the operation is a limit operation, and the aggregation engine is further configured to modify default operation of a sort operation within the plurality of data operations to minimize resources instantiated during execution of the sort operation.

24. The distributed database system of claim **22**, wherein the filter criteria specifies a limit on a number of returned results from the database.

25. The distributed database system of claim **14**, wherein the aggregation engine is further configured to determine a data architecture for the data stored on a database.

26. The distributed database system of claim **25**, wherein the aggregation engine is further configured to modify the default execution of at least one of the plurality of data operations responsive to the determined data architecture.

27. The distributed database system of claim **26**, wherein the act of determining the data architecture includes an act of determining the data architecture includes data shards, and the aggregation engine is further configured to modify the default execution of a sort operation to sort by merging data results returned from a plurality of the data shards.

* * * * *