Views and Transactional Storage for Large Graphs

Michael M. Lee, Indrajit Roy, Alvin Auyoung, Vanish Talwar, K. R. Jayaram, Yuanyuan Zhou

Views and Transactional Storage for Large Graphs

Michael M. Lee[1], Indrajit Roy[2], Alvin AuYoung[2], Vanish Talwar[2], K. R. Jayaram[2], and Yuanyuan Zhou[1]

[1] University of California, San Diego. {mmlee,yyzhou}@cs.ucsd.edu
[2] HP Labs {indrajitr,alvina,vanish.talwar,jayaramkr}@hp.com

Abstract. A growing number of applications store and analyze graph-structured data. These applications impose challenging infrastructure demands due to a need for scalable, high-throughput, and low-latency graph processing. Existing state-of-the-art storage systems and data processing systems are limited in at least one of these dimensions, and simply layering these technologies is inadequate.
We present Concerto, a graph store based on distributed, in-memory data structures. In addition to enabling efficient graph traversals by co-locating graph nodes and associated edges where possible, Concerto provides transactional updates while scaling to hundreds of nodes. Concerto introduces graph *views* to denote sub-graphs on which user-defined functions can be invoked. Using graph views, programmers can perform event-driven analysis and dynamically optimize application performance. Our results show that Concerto is significantly faster than in-memory MySQL, in-memory Neo4j, and GemFire for graph insertions as well as graph queries. We demonstrate the utility of Concerto's features in the design of two real-world applications: real-time incident impact analysis on a road network and targeted advertising in a social network.

Keywords: Graphs, transactions, views, event-driven analysis.

1 Introduction

Graph-processing applications are quickly emerging as a critical component in domains like social networks, road traffic, and biological networks where data exhibit natural graph structure. Building large-scale graph applications requires middleware support for storing data and for accelerating graph queries. Many of today's graph applications exhibit a need for high volume storage, low-latency updates, and interactive responsiveness. Individually, these requirements do not present a unique challenge, but taken together, they pose a significant challenge to both state-of-the-art storage and data-processing systems. We detail two of these emerging requirements, and how they translate into challenges for the supporting system infrastructure:

- **Scalability and consistency.** Because many of today's graph applications run on the critical path of an on-line workflow, a graph store needs to provide a combination of adequate query throughput and data ingestion rate. For example, Facebook receives more than 200,000 events per second [1] while Twitter ingests approximately 80 TB/day of new data [2]. However, without meaningful consistency semantics, such as transactional guarantees, writing distributed graph applications will be difficult and error prone.

- **Event-driven processing.** Some graph applications, such as those used by emergency technicians to respond to incidents, must be real-time to be useful [3]. For example, the California highway road sensors requires ingestion of new data every 30 seconds for over 26,000 sensors [4]. Such an application is largely event-driven based upon incidents (i.e., accidents, slowdowns) occurring on the road graph, and triggers computation to predict the spread and duration of the incidents. Supporting an API with flexible event processing on the graph store eases the development of these graph applications. Multiple applications monitoring similar events can avoid redundant client-side computation on event detection. Moreover, events can be used to monitor and dynamically optimize the store itself (such as graph layout) to further improve query performance.

1.1 Limitations of current systems

State-of-the-art solutions are not designed to address these challenges simultaneously. Traditional storage systems such as relational databases and NoSQL stores do not inherently retain the structure of the graph, and are unsuitable for computing graph algorithms [5]. As shown in Table 1, using a fast caching layer such as Memcached on top of a relational database can scale the performance of resolving graph queries. However, this approach relies on pre-computing the set of queries, and thus requires the workload to be known in advance, or returning a computation based upon stale data.

Distributed in-memory stores (GemFire [6]) provide dynamic scalability and high performance while also supporting transactions. However, similar to traditional relational databases, they do not provide native support for graph objects and hence are slow for graph queries.

Contemporary data-processing frameworks such as MapReduce [7], Pregel [5], Spark [8], or GraphLab [9] optimize for batch analysis by assuming data is largely read-only, and hence are ill-suited for concurrent read and write queries. Since many of these systems are designed for long running distributed graph analyses, the overhead from additional communication and setup costs may exceed the actual computation time for small graph queries [10].

In contrast, specialized graph databases perform complex graph queries quickly, and concurrently with graph updates [11,12,13,14,15,16]. These systems are, however, limited to a single machine (or a set of replicated images) and therefore do not scale with either the query rate, storage capacity, or data ingestion rate. Trinity [17], a distributed graph engine, lacks support for transactional storage of graph objects. Additionally, none of these graph databases support event-driven processing.

Technology	**Graph queries**	**Transactions**	**Event processing**	**Scalable**
RDBMS (MySQL, etc.)	Slow due to table joins	Yes	Yes, using triggers	No
RDBMS + memcached [18]	Fast but stale results	No	No	Yes
Distributed in-memory stores [6]	Slow	Yes	Yes, using triggers	Yes
Batch graph frameworks [5]	Fast but offline	No	No	Yes
Graph DBs (Neo4j [12], DEX [15])	Fast, online	Yes	No	No
Trinity [17]	Fast, online	No, explicit locks	No	Yes
Concerto Design choice →	**Fast, online** In-memory data structures	**Yes** Distributed transactions	**Yes** Graph views, notifications	**Yes** Distributed processing

Table 1. Comparison with competitive approaches. Concerto has multiple advantages over each system.

1.2 Contributions

We have designed Concerto, a graph store that preserves the functionality of specialized graph databases without sacrificing the ability to scale or build event-driven applications. As shown in Table 1, Concerto differs from existing work in its combination of two fundamental design principles.

First, Concerto provides distributed, *in-memory, transactional storage* of graph elements. Unlike traditional databases, Concerto embeds the graph structure within the aggregate memory of the cluster. While this layout incurs more storage than a traditional table-based layout, it enables otherwise expensive graph traversals to be performed quickly. Unlike existing graph databases, Concerto is designed to maintain data consistency across *multiple* servers using distributed transactions. The costs of the distributed transaction protocol are compensated by several performance optimizations: in-memory representation, fewer network roundtrips, and parallel computation.

Second, Concerto introduces the notion of *graph views*, which allows an application to denote subgraphs of interest. Applications can compose different views to form meaningful groups and run graph analysis on them. Applications can also register user-defined event handlers on a view. As we discuss in Section 6, some graph applications are naturally expressed as event-driven programs, and in our experience, the extensibility provided by Concerto improves performance and simplifies application programming. Moreover, views act as hints about which graph entities are related, thereby providing a means to enhance data migration, or partitioning policies [19,20,21] and reduce communication overhead.

We empirically compare Concerto against in-memory executions of MySQL, a standard relational database; Neo4j, an open-source graph database; and GemFire, a commercial in-memory distributed store. Concerto is 10× faster in bulk insertion throughput than Neo4j while consuming 3× less memory. Similarly, Concerto is more than 7× faster than MySQL for interactive k-hop query performance. Scaling results on 64 instances shows that Concerto can complete a 3-core computation of a 90-million node graph in only 12 minutes, compared to nearly 6 hours on 64 instances of GemFire. We also demonstrate Concerto's features through two real-world inspired applications: real-time incident impact analysis on a road network, and targeted advertising in a social network.

2 Graph storage

Concerto stores graph objects in memory and across distributed commodity servers in data centers (Figure 1a). A distributed shared memory implementation provides a global address space on which graph objects are allocated. Graph traversals take place on the distributed graph representation using server-side RPC calls batched for performance. Concerto's key contributions are in the *in-memory graph representation* and the use of *efficient, distributed transactions* to provide concurrent access and online data migration.

2.1 Graph representation

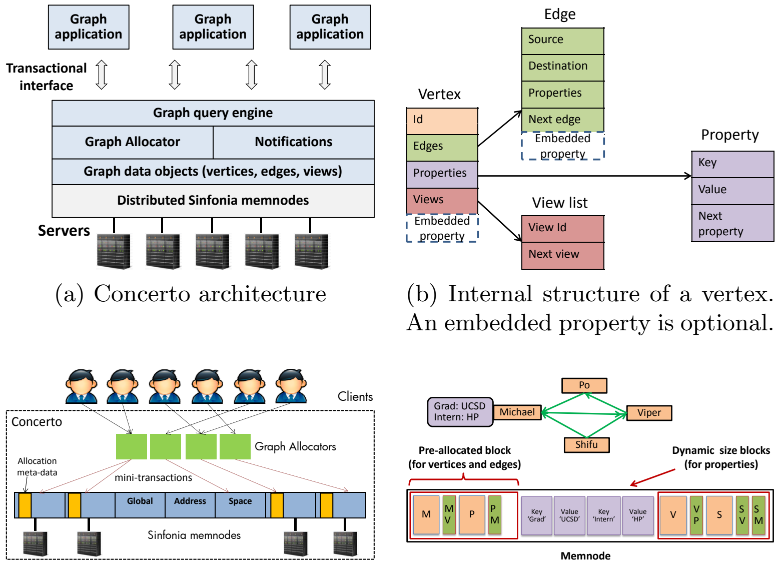
Concerto has three basic data types to store the application graph data: `vertex`, `edge`, and `property`. A `property` element contains attributes and can be attached to a vertex or edge. Vertices and edges can have multiple properties. Concerto exposes APIs to graph applications to create and update the above graph elements. New graph objects are allocated on a global address space provided by Sinfonia [22], a distributed shared memory system. Sinfonia exposes a flat memory region per-server called memnode which are combined to create a single global address space.

Concerto stores the logical graph using a layout optimized for in-memory reads and inserts. As shown in Figure 1b, vertices, edges, and properties are represented as records with pointers. A vertex has a pointer to a list of its outgoing edges. An edge has pointers to its source and destination vertices and to the next edge of the source vertex. Thus, all outgoing edges of a vertex can be accessed consecutively starting from the first edge. Co-locating vertices and edges in contiguous blocks of memory, and storing pointers to related graph objects allow graph traversals to be performed quickly at the cost of additional storage. Similar to edges, properties are chained together as a list. Both vertex and edge records point to the head of their property lists.

In Concerto, each vertex and edge is a fixed-size record while properties can be of variable size. Using an appropriate fixed size, a vertex or edge can be retrieved in one read transaction (one network roundtrip between a client and a Concerto server) as both the address and size of the data are known in advance. However, accessing properties of a vertex or edge may require more than one transaction. First, the vertex has to be read to determine the address of the property and then the property is read in the next transaction. In some applications, certain properties are accessed often. To retrieve these frequently accessed objects in one read transaction, properties can optionally be *embedded* in the vertex or edge records. Figure 1b depicts embedded properties attached to vertices and edges.

2.2 Use of transactions

Concerto uses distributed transactions to provide consistency and concurrency for graph allocation, access, and updates. Unlike simple key-value data, graph



(a) Concerto architecture

(b) Internal structure of a vertex. An embedded property is optional.

(c) Distributed graph allocation.

(d) Example graph layout in *memnode*; vertices and edges are referred to by their first letters.

Fig. 1. Overview of Concerto

data can seldom be partitioned into shared nothing regions, and hence we need to support transactions that occur *across* machines. To balance consistency with efficiency, Concerto leverages a distributed compare-and-swap primitive called a *mini-transaction* provided by Sinfonia to support such distributed transactions. Mini-transactions are a performance-optimized implementation of the two-phase commit protocol. Concerto also provides other optimizations to minimize the number of transactions used. These include batching graph operations during traversals and reducing the number of indirections for graph object access. Using these optimizations, Concerto can, in the common case, perform reads of vertices, edges or attributes in a single network roundtrip, and finish writes in two network roundtrips. By comparison, transactionally updating even a single value in GemFire requires at least three network roundtrips. Below, we discuss examples of how transactions are used in Concerto.

Graph allocation. During allocation of new graph elements (e.g., `vertex`, `edge`) it is important to ensure a unique address is assigned to the graph element even if two concurrent users request memory. Concerto uses transactions to achieve this. As shown in Figure 1c, whenever an allocation request is received, the Concerto graph allocator contacts the Sinfonia memnode. Upon allocation of an address space, an entry is made to the allocation meta-data on the memnode. Concerto wraps these operations in transactions which ensures that the meta-data for the allocator remains consistent during concurrent allocation requests. Note that the use of transactions to allocate and manage each element incurs overhead, especially for vertices and edges which are only a few tens of bytes. To reduce this, Concerto pre-allocates large memory blocks from memnodes and appends new vertices and edges until the block fills up. Pre-allocated blocks reduce the amount of meta-data stored on memnodes, and also the number of network roundtrips (and possible write conflicts) from allocation requests. Figure 1d illustrates how pre-allocated blocks store vertices and edges.

Graph updates. Transactions are also used to allow in-place updates to existing graph elements with other (concurrent) accesses. Internally, the Concerto transaction API calls the Sinfonia mini-transaction subsystem which allows updates to be made to graph elements on distributed machines (in this case source and destination vertices).

Graph partitioning. Concerto uses transactions to provide *online data migration* for an application to optimize a graph partition. This can be used, for example, when adding or removing servers, or when handling data hotspots. Table 2 shows the three migrate functions available to applications. These functions implement migration as a series of tasks wrapped inside distributed transactions. For example, when migrating a vertex, the vertex and its associated data are copied to the new server, the original copy is deleted, and all incoming pointers to the vertex are updated. These tasks happen inside a transaction during which time other non-conflicting operations can continue concurrently.

Function	**Description**
migrateVertex(V, s)	Move V and its data to server s
migrateView(View, s)	Move view elements to server s
migrateGraph(View, map)	Move elements based on map

Table 2. Functions to migrate data.

3 Graph Views

The primary programming innovation of Concerto is the notion of application-specific event processing using *graph views*. The concept of views is well-studied in the database literature. Concerto extends this concept to distributed graph stores. In Concerto, a view is a subgraph of interest on which applications can run graph algorithms and also register generic event handlers. By using event handlers, an application is easily expressed as an event-driven program.

3.1 Programming model

Concerto provides a `View` class to create and manage graph views. Views are subgraphs and comprise of a list of vertices, edges, and properties. Views are primarily created to isolate regions of interest and can constrain a query to execute only on a subgraph. For example, Concerto applications use the BSP programming model to implement distributed graph algorithms by specifying code that runs on each vertex [5,23]. Graph views provide application-specific

Function	Description

onReadVertex(V)
onReadEdge(E)
onReadProperty(P)
	Invoked on read operation. Passes element where read occurred.

onUpdateVertex(V)
onUpdateEdge(E)
onUpdateProperty(P, val)
	Invoked on write operation. Passes element where write occurred. Old value of property also passed.

Table 3. `View` API for event-driven processing.

semantics to these algorithms: applications can compose multiple views and then execute a distributed algorithm on the complex view. Consider the example mentioned in the introduction where the graph `G` represents a road traffic network and the graph application performs real-time accident impact analysis [3] on `G`. The application developer might create a view `P` and `M` corresponding to the cities of Palo Alto and Mountain View, respectively. If an accident occurs in Palo Alto, then the application can localize the execution of its impact analysis algorithm on `P` and demarcate the affected region `I=impactAnalysis(P)`. Now, the application can be easily extended to provide useful functionality; to find the best path from a location in Mountain View to Palo Alto, while avoiding the impacted region, a user would simply run a shortest path algorithm on the composed view:`(P - I)∪M`. Concerto supports basic set operations, such as union, intersection, and subtraction, on views. For example, two views can be merged or their common elements subtracted.

Views simplify the support for event-driven processing. Applications can define a view upon which to register event handlers. The View API (Table 3) can be invoked when read or write events occur in the subgraph. For example, `onreadVertex` function can be invoked when a read event occurs on a vertex in the view, and `onUpdateProperty` function is invoked on a write event to a property element in the view. To implement function invocation, the view pointers are stored in graph elements (Figure 1b). Specifically, whenever a read or an write occurs on a graph element, the view pointer(s) associated with the graph element are traversed and the corresponding function is invoked.

Applications can invoke custom code using the View API. In our experience, read functions are broadly useful for gathering statistics and for monitoring the store. For example, the `onReadVertex` function can be overwritten to determine whether too many clients are reading the view members. By monitoring read throughput, data may be migrated proactively to reduce hotspots. Handlers for write events may benefit from more customization. For example, in our road traffic application, write events might specify the location of an accident, which would trigger execution of the traffic impact analysis in that region.

3.2 Data structures

Supporting graph views requires a trade-off between compact storage and fast subgraph traversal. For fast graph traversals, the ideal approach makes a copy of the subgraph corresponding to a view, enabling traversals to occur directly on the subgraph. However, this approach has serious shortcomings. For large graphs, applications may create hundreds or thousands of views. Some of these views may overlap and store (possibly) large, redundant portions of the original graph. Therefore, this approach may lead to unnecessary space explosion from duplicate copies of nodes and edges. Additionally, when updates occur, preserving the structural consistency across the views and the original graph will result in significant overhead.

To overcome these problems, the View class only stores the identifiers of its members (vertices and edges), and a scratch space to store properties about the view itself. Therefore, views store only membership information and not structural information. This storage format has the singular advantage of low space overhead. However the compact representation has the unfortunate side-effect that by looking at only the internal representation of a view, it is not possible to traverse the subgraph. For example, the view may not contain enough information to determine the neighbors of a certain vertex. Instead, the view's stored information has to be combined with the actual graph to traverse the subgraph contained in the view.

Concerto uses hash maps to speed up traversals on a view's subgraph. Vertex and edge identifiers are hashed for fast lookups. To execute a graph algorithm on a view, the application specifies the code that runs at each vertex, but Concerto ensures that the algorithm will be constrained to only the view members.

3.3 Event-driven processing

Supporting event processing in a graph store raises several questions. Since events on a graph can span different servers, how should the graph store aggregate such information? Intercepting each event may introduce undesirable processing overhead. If so, how do we prevent event processing from unduly impacting the query throughput of the graph store?

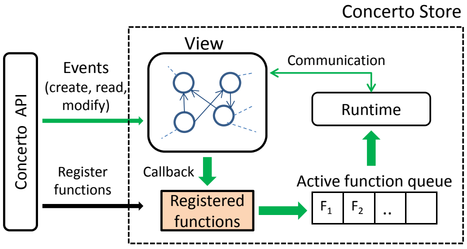


Fig. 2. Control flow of event processing.

Views store the functions to be invoked when specified events occur. For example, views map the six function names in Table 3, such as `onUpdateVertex`, to the programmer-specified functions. Whenever events occur in a view, the runtime invokes the corresponding functions. Applications register functions to a view by calling `register()` on the the view. For example, `V.register( onUpdateVertex, myFunc)` will register the function `myFunc`. Concerto will invoke this function whenever any vertex is updated in the view `V`. Internally, the function is stored as an executable.

When the events of interest, e.g., vertex updates, occur on a graph element, the Concerto runtime needs to determine which function registered with the view should be invoked. As explained in Section 2, view members, such as vertices, have a reverse pointer to their view object. This pointer is used to reach the functions that need to be executed after the event occurs at the graph element. Figure 2 illustrates the control flow during event-driven processing. The invoked functions are first appended to a queue. These queued functions are executed by a dedicated thread pool (separate from those handling queries) and, hence, provide coarse-grain performance isolation between queries and event processing. During execution, these functions can use the Concerto runtime for read access to elements of the view. For example, after a traffic accident the impact analysis function may traverse the vertices in the view to determine the affected region. The invoked functions can also store persistent data in the property fields of the view for subsequent processing. For example, a monitoring function may store statistics about reads and writes as a property of the view.

Concerto can leverage off-the-shelf publish-subscribe systems for large-scale event propagation. We believe our work to be complementary to these systems.

4 Fault Tolerance and Security

Concerto simplifies the graph store architecture by delegating most of the fault recovery mechanisms to Sinfonia. Sinfonia provides atomicity, consistency, isolation, durability (ACID), and availability if replication is enabled. These guarantees are independent of client-failures and the size of the graph. This design choice ensures that the graph store can easily be ported to other platforms such as distributed key-value stores. The Concerto prototype uses Sinfonia's disk-logging mechanisms to recover from memnode failures.

Sinfonia's fault tolerant global address space implies that data stored in Concerto is recoverable. However, we need mechanisms in Concerto to regain consistency (upon recovery) of the graph allocators. Graph allocators store all their meta-data in the memnodes. However, if a graph allocator fails then some of the memory may be leaked (e.g., pre-allocated blocks may be left dangling). The recovery process in Concerto goes through the allocator meta-data in each memnode and entrusts any dangling memory block to an active graph allocator.

Unlike data operations, event processing in Concerto's current prototype is not completely fault-tolerant. The difference in guarantees occurs because events are processed asynchronously to isolate query performance from event processing. As a result, an untimely fault can result in lost events. For example, when an update occurs, the write operation may return results to the client even though the triggered event processing code may still be executing a computation. If there is a fault before the update operation returns, then Concerto's recovery process will ensure that both the write operation and the event-processing code is correctly re-executed (or the client is notified of the failure and can retry). However, if a fault occurs after the update operation completes but before the event processing code completes, then the event may be lost. One can make event-processing fault tolerant by using a fault-tolerant message queue which we plan as future work.

Concerto assumes that functions registered with views are written by trusted applications. Malicious code can impact both the graph store and the stored data. The current prototype does not provide additional security features to constrain malicious functions. In the future, standard security techniques, such as sandboxes, may be used to limit the power of these functions [24]. Also, eventual consistency will let Concerto scale to more servers, improve its performance via asynchronous updates, and may decrease the latency of graph operations. However, eventual consistency is difficult to reason about and program.

5 Evaluation

Concerto consists of approximately 4, 500 lines of C++ code for distributed data allocation, query API, distributed graph traversals, and event processing. These lines of code do not include the Sinfonia codebase.

We compare Concerto against MySQL, a well known relational database; against Neo4j, a commercial graph database; and GemFire [6], a commercial, in-memory distributed data management platform. GemFire uses hashing to store data in memory regions distributed over multiple nodes and provides a SQL-like interface. All experiments are performed using a cluster of 100 HP SL390 servers running Ubuntu 11.04. Each server has two Intel Xeon X5650 processors (total of twelve 2.67GHz cores), 96GB of DRAM, 120GB SSD drives, and 10 Gbps NIC. In some cases, where the 96 GB memory limit was exceeded, we ran Neo4j on a separate 1 TB memory server, however 96 GB was never approached for a single Concerto memnode in these graphs.

In our experiments, all systems run *in-memory*: Neo4j is run on a `ramfs` partition, MySQL uses its memory engine, and Concerto is run without replication. GemFire is run with one logical data region, distributed over multiple nodes (the number of nodes is specified in each experiment). In MySQL and GemFire, the graph is stored as a table of edges. We optimize MySQL query performance by using B-tree and hash indexes (GemFire uses hash maps). Workload generators are located on different servers from those hosting the store.

From our evaluation, we find that:

- Concerto is *fast*. It can ingest millions of vertices and edges per second and is more than 25× faster than other systems for k-core on large graphs and uses 3× less memory than Neo4j.
- Concerto's performance *scales* with the additional servers. It can calculate the 3-core on a 90 million vertex graph in less than 12 minutes on 64 instances compared to 45 minutes on a single memnode. The same computation takes more than 6 hours by GemFire.
- Concerto's graph views provide *extensibility*. Due to views and event-processing, Concerto's implementation of a road traffic application is 10× faster than a poll based system.

Table 4 describes the graphs used in our experiments. For example, *Twitter-L* represents 51 million users with 2 billion follower relationships that was collected from the Twitter Web site. The *Social-S/L* graphs represent synthetic social network graphs generated using the model proposed by Newman [25].

Graph	Vertices	Edges	File size	Experiments
Twitter-S	33M	282M	6.5GB	Insert, k-hop, monitoring
Twitter-L	51M	2B	38GB	
Social-S	3M	13M	197MB	Insert, k-core
Social-L	90M	405M	7.5GB	
Road-CA [26]	2M	5M	84MB	Traffic analysis

Table 4. Graphs used in different experiments.

Inserts/sec	Vertex	Edge	Vertex(bulk)	Edge(bulk)
Neo4j	282	337	6,120	6,467
MySQL	21,898	15,457	504,209	324,352
GemFire-1	5,234	6,001	153,245	165,324
Concerto-1	6,584	7,089	1.1 million	0.9 million
GemFire-10	27,512	23,092	1.3 million	1.0 million
Concerto-10	29,695	27,122	2.6 million	1.8 million

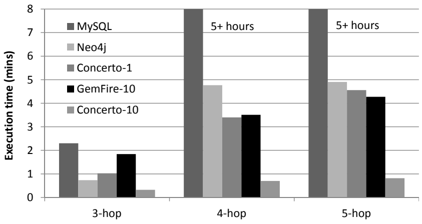
Table 5. Comparison of insertion throughput. Concerto/GemFire-1,10 represent running with 1 and 10 instances, respectively.

5.1 Performance results

We first compare the insertion throughput and query latency of Concerto, Neo4j, GemFire, and MySQL.

Insertion throughput. Table 5 compares how many vertex and edge elements can be inserted per second by the different stores for the Twitter-S graph. We also compare bulk loading all the vertices and edges of the Twitter-S graph for the different stores. Bulk loading avoids overhead from multiple memory allocations by inserting vertices and edges in one request. Insertion requests are issued from multiple clients to maximize the throughput. Our results show that a single instance Concerto is more than 21× faster than Neo4j. For example, Concerto can insert more than 6,500 vertices/second compared to only 282 vertices/second for Neo4j. Inserting single data items into MySQL is faster than a single-server instance of Concerto and GemFire because MySQL is highly tuned and stores the graph in a simple table format. However, as we discuss in the next section, this table representation considerably limits graph query performance in MySQL. GemFire and Concerto exhibit similar performance with single-item insertions, but Concerto is $1.8 - 7\times$ faster than GemFire with bulk insertions. Similar to single-insertion, GemFire still needs to hash every element in the bulk insertion case resulting in lower performance. Concerto can parallelize ingestion to increase throughput. With 10 instances, Concerto inserts approximately 2.6 million vertices/second and 1.8 million edges/second. Scaling ingestion throughput is particularly useful for applications that must load very large graphs.

Graph query: k-hop. A common query in many graph applications is to retrieve a vertex and its neighbors that are k-hop distance away. Figure 3 compares the latency of retrieving upto 5-hop neighbors of a vertex in the different systems. For a 1-hop distance, MySQL and Concerto have similar performance, while Neo4j is noticeably slower. Neo4j's Java implementation is the main reason for the slowdown. However, for queries requiring more than 2-hops, MySQL is the slowest. This result is not surprising because for each hop, it has to perform a join operation, which is known to be an expensive operation for a database. For fewer



(a)

System	1-hop (ms)	2-hop (ms)
MySQL (in-memory)	9	303
Neo4j (in-memory)	42	1,320
Concerto-1	2	340
GemFire-10	10	295
Concerto-10	2	265

(b)

Fig. 3. K-hop latency. For clarity, 1 and 2 hop results are in the table.

than 3-hops, the overhead from two table join operations is insignificant due to use of the MySQL index. On the other hand, graph databases are optimized for such larger traversals. For greater than 2-hops, both Neo4j and Concerto-1 are at least 2-100× faster than MySQL, with the speedup increasing with the number of hops in the query. GemFire-1 performs worse than MySQL and hence we show only the GemFire-10 numbers. While Neo4j exhibits similar or better performance than the single-server instantiation of Concerto, Concerto-10 is 2-6× faster than Neo4j and and 5× faster than GemFire-10.

System	Social-S	Social-L
Neo4j	6 min	64 hrs
GemFire-1	5 min	25 hrs
MySQL	4 min	22 hrs
Concerto-1	1 min	45 min

Fig. 4. 3-core execution time.

Graph algorithm: k-core. The k-core of a graph determines the subgraph where each vertex has at least k neighbors on the induced subgraph. Vertices with a larger "coreness" value (i.e. k) correspond to nodes with a more central position in the network structure [27]. The k-core of a graph is obtained by recursively deleting vertices with degree less than k, until the degrees of remaining vertices is larger than or equal to k.

For Concerto, we implement the parallel k-core decomposition algorithm [27]. Table 4 shows the time taken by different systems to calculate the 3-core of two social graphs. For the 3 million vertex Social-S graph, MySQL, GemFire and Neo4j perform similarly, computing the 3-core of the graph in 4, 5 and 6 minutes respectively. Concerto, however, computes the 3-core much faster, requiring only 1 minute. For the 90 million vertex Social-L graph, Neo4j requires over two days to compute the 3-core, whereas MySQL and GemFire complete the same computation in 22 and 25 hours, respectively. Concerto completes the computation in only 45 minutes.

To understand these numbers, we observe that each round of the k-core decomposition consists primarily of two phases: a graph scan to find the vertices that need to be deleted, and a short traversal of each vertex to find their degree in the induced subgraph. The bottleneck in Neo4j is the part of the algorithm that must perform a scan of all the data; such scans are known to be slow in Neo4j for large graphs. MySQL's and GemFire's advantage in scans and predicate evaluation (using indexes) is the primary reason for the speedup relative to

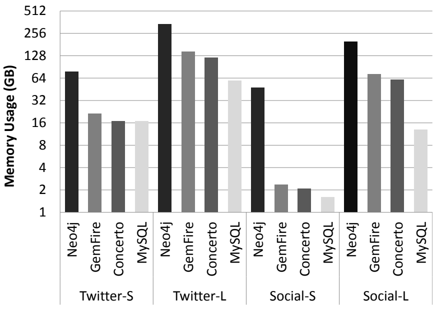


Fig. 5. Comparison of memory usage.

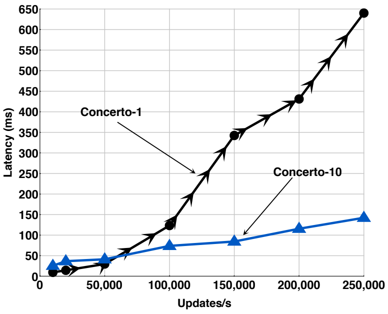


Fig. 6. View update latency

Neo4j for large datasets. We note that for the Social-L graph we used a hash index since MySQL could not create such a large B-Tree index due to a known unresolved bug in its implementation [3]. In contrast, the design of Concerto allows it to perform both graph scans and traversals quickly. It calculates the 3-core in approximately 45 minutes, which is 29× faster than MySQL, 33× faster than GemFire, and 85× faster than Neo4j.

5.2 Memory footprint

In Figure 5, we compare the storage footprint of each system, which, in this case, is entirely in memory. For MySQL we quote the numbers when only a hash index is created, which is much more memory efficient than creating a B-Tree index. Over all data-sets, MySQL has the smallest storage footprint as it stores only the edge information in the form of a table. Concerto requires $1.3 - 4.7\times$ more storage than MySQL, and requires similar storage as GemFire but is $2.8 - 22.7\times$ more space efficient compared to Neo4j. GemFire consumes less memory than Neo4j due to optimizations in object serialization and deserialization – only cached objects exist in deserialized forms, while remaining objects exist in smaller, serialized form. Apart from the overhead of Java, Neo4j also stores extra metadata and hence consumes more memory than Concerto. For example, in Neo4j the outgoing edges of a vertex are stored in a doubly linked list which incurs the additional cost of a back pointer per edge.

Revisiting the performance numbers from k-hop and k-core in the context of memory footprint reveals that for simple queries (e.g., 1-hop or 3-core) over small data sets, running in-memory MySQL offers roughly the same performance-vs-memory trade-off as single-server Concerto and a better trade-off than Concerto-10. However, as the data set size or query complexity increases, the additional performance improvement of Concerto outweighs the additional storage requirement. For example, when running k-core, single-server Concerto requires 4.7× more storage than MySQL, but improves k-core latency by a factor of 29×. In

[3] http://bugs.mysql.com/bug.php?id=44138

the case of Neo4j, this difference is even more pronounced, where Concerto requires 3.3× *less* storage, and yet improves k-core latency by a factor of 85×. Compared to GemFire, Concerto consumes a similar amount of memory but is 33× faster on k-core.

5.3 View updates

Figure 6 is a microbenchmark to measure the latency in processing view updates. We created a view on the Twitter-S graph such that 20% of the vertices are part of the view (around 7M). We use a client to send randomly generated updates to vertices both within and outside the view . Whenever a vertex is updated in the view, we use event-processing to increment the count of writes occurring on the view. The Y-axis in the plot shows the time interval between a vertex update and the completion of the event handler. In Concerto-1, the latency increases substantially as the update rate becomes more than 50K/s. The increased latency is because the single server reaches full capacity utilization and incurs queuing delay. In Concerto-10, the view update latency is higher than Concerto-1 initially, due to the network communication to update the view statistic that resides at one server. However, since the graph is distributed across multiple nodes, Concerto-10 can handle a higher update rate. The average delay is under 150ms for Concerto-10 even when the update rate is 250K updates/s. As a reference point for update rates, Twitter and Facebook receive 100K-200K update events per second [1,2].

5.4 Scalability results

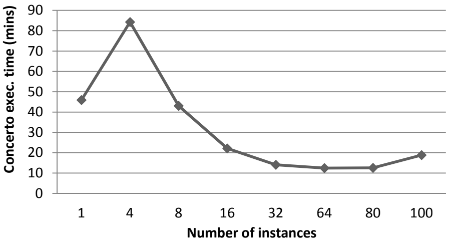
Unlike Neo4j, Concerto can leverage distributed parallelism to improve performance. Figure 7 shows the effect of scaling on the execution time of 3-core on the Social-L graph. As we increase the number of Concerto instances to 64, the execution time drops from 45 minutes to 12 minutes. At four Concerto instances the execution time is higher than the single server case because of the extra communication required. Similarly, beyond 80 instances the communication overhead for this dataset overshadows the benefit of increasing parallelism. The table in Figure 7 shows that GemFire's performance improves with more instances. However, even at 64 instances it still requires 6 hours to complete.

6 Case Studies with Views

In this section, we consider how views and event-processing in Concerto can ease development and improve performance of real-world inspired applications.

6.1 Real-time traffic impact analysis

The California Performance Measurement System (PeMS) is a network of road sensors spanning the major metropolitan freeways in California; these 26,000



#Instances	1	4	8	16	32	64
GemFire (Time in hrs)	25	16	13	10	9	6

Fig. 7. Distributed k-core: Concerto, GemFire execution time. (Social-L)

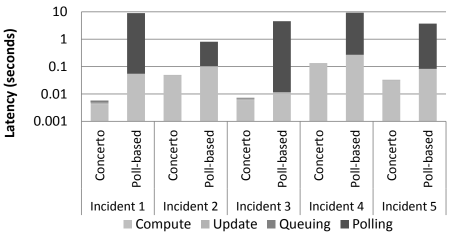


Fig. 8. End-to-end latency in finding incident impact region: Comparison between Concerto and a polling based system. Lower is better.

sensors collect data every 30 seconds, generating over 2 GB of data each day [4]. The primary challenge isn't the scale of the data, but the *real-time* nature of the application. We revisit the example described in sub-section 3.1 and implement an application considered by both Kwon et al., and Miller et al: a statistical technique to estimate the time and spatial impact of a road incident (e.g., an accident, obstruction) on surrounding traffic [3,28]. When any such incident occurs, the application needs to react by analyzing the road network graph to predict the impact region of this incident, and possible re-calculated the shortest path between two endpoints of an impacted commute. Low latency is of the essence in order to notify the appropriate authorities to respond [3].

The application leverages Concerto in three ways. First, road sensors are stored as vertices, and connecting road segments are stored as associated edges in a graph. Each vertex contains information collected by its associated sensor (i.e., traffic flow, velocity, external incidents, which are uploaded from external sources). Second, a specific region of interest – for example, a municipality – forms a graph view such that the relevant client can run analysis when events occur. Finally, a function is registered with each view to run the impact analysis algorithm upon occurrence of an incident. The analysis function can use the information contained in the sensors that span the view.

We construct a graph based on the California road network [26], and generate 10 independent views (non-overlapping subgraphs) of size 2000 to approximate independent municipalities. We drive the experiment using synthetic traffic and incident data over a 25-minute window; this data is drawn from a distribution sampled by historical PeMS data, and approximately matches the findings of Miller, et al [3]. Therefore, at every 30-second interval, a traffic update invokes an `updateProperty(sensorID,sensorData)` in Concerto, and when an incident occurs (also associated with a specific sensor), `updateProperty(sensorID, incident)` is invoked. Upon completing the incident impact analysis, a shortest path between a fixed source and destination municipality is recalculated, with any impacted subgraph removed from the calculation.

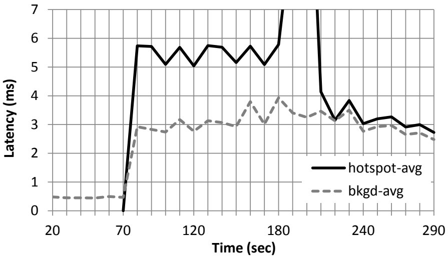


Fig. 9. Request latency observed during a hotspot and migration (clipped points represent downtime).

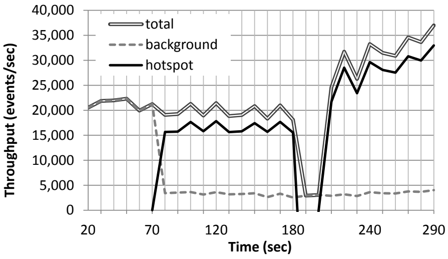


Fig. 10. Throughput observed during a hotspot and migration.

Figure 8 shows the end-to-end latency of determining the impact region for the first five incidents. To measure the performance benefit of using Concerto, we compare the latency of implementing this application in Concerto without event-driven processing, thus requiring polling and client side processing (*poll-based* in Figure 8). We show a breakdown of the time it takes to compute the impacted region, update the incident sensor, and from queueing delay, and polling overhead. We use a 10-second polling interval, which is close to the time taken to scan all the vertices in the ten views and read whether an incident has occurred. In Figure 8 we see that Concerto can find the impacted region in less than 100 milliseconds, largely due to server-side processing in both the event handling and code execution. The polling-based system takes from 1 to 10 seconds, and thus is slower by upto two orders of magnitude from Concerto. Even discounting the polling overhead, the poll-based system takes one second to complete because of the costly client-side graph traversal.

6.2 Hotspot migration

Large social networks such as Facebook expose their infrastructure to third-party advertisers wishing to target particular users (e.g, through Facebook Ads API [29]). An increase in targeted advertising usually coincides with increases in traffic from trending topics or external events that impact the social graph (e.g., the Super Bowl). This rapid increase in traffic can cause a workload hotspot, especially if members of the view are co-located on the same set of physical servers. Concerto can dynamically load balance data corresponding to a view to mitigate such workload hotspots. To demonstrate these features, we replay a synthetic trace of read and write traffic on the Twitter-S graph stored across three Concerto instances. To simulate peak load, we create a view (called *hotspot*) by randomly selecting $1,000$ Twitter users (these correspond to vertices in the Twitter-S graph) stored in the same Concerto sever. We assign a designated set of clients to this view to simulate heavy-hitters and continuously send requests to those vertices. Concerto handles the heavy hitters by using a migration policy such that the view members are evenly distributed across the three Concerto servers. This policy is implemented by gathering statistics of the number of requests hitting the view and when it exceeds a threshold (i.e hotspot occurs), the data migration policy is invoked. All of this is done using the View event-driven processing API described in Section 3.1. Figure 9 shows the timeline for the above scenario. The hotspot occurs at time 70 seconds, at that time the average request latency seen by the heavy hitters (*hotspot-avg*) increases to 6ms compared to less than a millisecond initially. The average latency of other clients in the store also increases (*bkgd-avg*) as some of their queries are on the view data. At time 180 seconds, the migration starts and moves approximately one-third of the view members to the remaining servers. During migration 683 vertices have to be moved which requires updating $10,916$ edges. The total migration time takes approximately 19.3 seconds, representing a downtime window during which the heavy hitters cannot access the store. Note that our migration implementation isn't well optimized and the migration time of 19.3 seconds is on the higher side which can be reduced. At the end of the migration, the average latency for both the heavy hitters and the other users becomes the same. Also, as shown in Figure 10, after the migration, the store can handle more traffic from the heavy hitters as the data is now spread across more servers. The effect of this migration is reflected by the increase in the total throughput beyond time 200 seconds. Online data migration can also be used to optimize query performance in other cases. For example, in another experiment, we observe that executing 3-core on the 3M vertex graph (Social-S) is 1.6× better running on 10 Concerto servers than executing it on 32 instances. Due to the communication overhead, it takes 24 seconds to calculate 3-core on 32 instances compared to less than 15 seconds with only 10 instances. Therefore, in this case, the social media application can move the data of a view to span fewer machines before running the k-core query. Note that Concerto does not automatically partition the graph for optimal performance. But applications can use known partitioning schemes and register the partitioning logic with Concerto for dynamic data partitioning.

7 Related Work

Relational databases. Common graph queries such as shortest-path and k-hop are both difficult to express and inefficient to implement in a relational model. Because these queries cannot be completed quickly enough to support interactive or real-time responses. Web 2.0 sites such as Facebook, Flickr, and Wikipedia complement their SQL-based backing store with an in-memory cache such as Memcached [18] to provide low-latency response. Unfortunately the need to use a caching layer to achieve performance scalability comes at the cost of relaxed transactional semantics, a severely restricted set of supported graph queries, and lack of extensibility using application defined functions.

Distributed datastores. GemFire [6] is the most closely related system to Concerto in that it is a scalable datastore that supports parallel query processing, event-driven processing and transactions. It exports an SQL-like query interface on top of a distributed key-value storage layer. It is capable of performing dynamic load balancing, event-processing over data, and in-memory caching of data objects across servers. Like relational databases and key-value stores, however, it does not provide explicit support for graph objects, thereby making graph queries inefficient.

Batch analysis. Existing approaches to large graph analysis focus on optimizing offline computation. Systems like Pregel [5], GraphLab [9], Horton [30] and algorithmic methodologies in the high-performance community [31,32] primarily address the challenge of scaling computation with the size of graph data, generally on the order of billions of nodes and edges. As a result, these domains restrict themselves to immutable, read-only data. On the other hand, Concerto is designed to address the challenge of providing low-latency computation with transactional semantics for complex graph problems where data is continuously ingested and modified.

Graph databases. Many specialized graph databases provide transactional guarantees and are optimized for typical graph operations, but largely do not scale storage [12,14,15] or storage or computation [11,16] to multiple servers. Kineograph [33] and Trinity [17] are the most closely related graph projects to Concerto, but do not provide semantics for user-defined functions or event-based, active computation. Kineograph is designed to provide transactional support for real-time graph updates in a distributed graph storage system; however, it does not explicitly support fast graph computations and instead stores graph elements using hash-based partitioning across graph storage nodes. Trinity does not provide transactional storage, and makes a different trade-off with how edges are named and represented in the graph. In the case of InfiniteGraph, no detailed technical documentation is available to provide a more informed comparison.

Graph views. Gutierrez et. al. [34] proposed database graph views as an abstraction mechanism on relational and object oriented databases. Their work includes derivation operators such as union, intersection, difference to define new graph views. However, their work is in the context of traditional databases and they do not provide a specific implementation. Concerto's view mechanisms build upon this prior work and provide a specific implementation in a distributed environment for non-relational, in-memory graph stores.

8 Conclusion

Many emerging applications require both scalable, transactional data storage, and interactive, low-latency graph analysis. Concerto is a distributed graph store that fills the gap between tiered database systems that scale, but perform poorly on graph queries, and recent graph frameworks which can efficiently compute graph algorithms but are offline and don't provide transactional storage semantics. Concerto's abstraction of graph views simplifies how graph applications are expressed, and provides mechanisms that can sustain update rates reported by Twitter and Facebook.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. Part of this research was sponsored by the DARPA GRAPHS program (BAA-12-01).

References

1. Facebook's new realtime analytics system: Hbase to process 20 billion events per day. http://highscalability.com/blog/2011/3/22/facebooks-new-realtime-analytics-system-hbase-to-process-20.html
2. Twitter by the numbers. http://mehack.com/twitter-by-the-numbers
3. Miller, M., Gupta, C., Wang, Y.: An empirical analysis of the impact of incidents on freeway traffic. Research paper HPL-2011-134, Hewlett Packard, Palo Alto, CA, USA (2011)
4. Caltrans performance measurement system (pems). http://pems.dot.ca.gov/
5. Malewicz, G., Austern, M.H., Bik, A.J., Dehñert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of SIGMOD. (2010) 135–146
6. GemFire: Technical white paper, copyright 2005 by gemstone systems. http://community.gemstone.com/display/gemfire60/EDF+Technical+White+Paper
7. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proceedings of OSDI 2004. (December 2004) 137–150
8. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of NSDI, San Jose, CA (2012) 1–14
9. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Proceedings of OSDI, Hollywood (October 2012) 1–14
10. Lattanzi, S., Moseley, B., Suri, S., Vassilvitskii, S.: Filtering: a method for solving graph problems in mapreduce. In: Proceedings of SPAA. (2011) 85–94
11. Infinitegraph: The distributed graph database. http://www.infinitegraph.com/
12. Neo4j: Nosql for the enterprise. http://neo4j.org/
13. Twitter flockdb. http://engineering.twitter.com/2010/05/introducing-flockdb.html
14. Iordanov, B.: Hypergraphdb: a generalized graph database. In: Proceedings of WAIM, Berlin, Heidelberg, Springer-Verlag (2010) 25–36
15. Martínez-Bazan, N., Gómez-Villamor, S., Escale-Claveras, F.: Dex: A high-performance graph database management system. In: Proceedings of IEEE ICDE Workshop on Graph Data Management, IEEE (2011) 124–127
16. Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., Haridasan, M.: Managing large graphs on multi-cores with graph awareness. In: Proceedings of USENIX ATC, Berkeley, CA, USA (2012) 1–12
17. Shao, B., Wang, H., Li, Y.: Trinity: A distributed graph engine on a memory cloud. In: Proceedings of SIGMOD. (2013)
18. Fitzpatrick, B.: Distributed caching with memcached. Linux Journal **2004**(124) 5
19. Huang, J., Abadi, D.J., Ren, K.: Scalable sparql querying of large rdf graphs. (August 2011) 1123–1134

20. Karypis, G., Kumar, V.: Metis - unstructured graph partitioning and sparse matrix ordering system. Technical report, University of Minnesota (1995)
21. Mondal, J., Deshpande, A.: Managing Large Dynamic Graphs Efficiently. In: Proceedings of SIGMOD. (2012) 145–156
22. Aguilera, M.K., Merchant, A., Shah, M.A., Veitch, A.C., Karamanolis, C.T.: Sinfonia: A new paradigm for building scalable distributed systems. ACM Trans. Comput. Syst. **27**(3) (2009) 5:1–5:48
23. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33** (1990) 103–111
24. Geambasu, R., Levy, A.A., Kohno, T., Krishnamurthy, A., Levy, H.M.: Comet : An active distributed key-value store. In: Proceedings of OSDI. (2010) 1–13
25. Newman, M.E.J., Watts, D.J., Strogatz, S.H.: Random graph models of social networks. Proceedings of the National Academy of Sciences of the United States of America **99** (2002) 2566–2572
26. Stanford large network dataset collection. http://snap.stanford.edu/data/index.html
27. Montresor, A., De Pellegrini, F., Miorandi, D.: Distributed k-core decomposition. In: Proceedings of PODC. (2011) 207–208
28. Kwon, J., Mauch, M., Varaiya, P.: The components of congestion: delay from incidents, special events, lane closures, weather, potential ramp metering gain, and demand. In: Proceedings of the TRB 85th Annual Meeting. (2006)
29. Facebook developers: custom audience targeting. https://developers.facebook.com/docs/reference/ads-api/custom-audience-targeting/
30. Sarwat, M., Elnikety, S., He, Y., Kliot, G.: Horton: Online query execution engine for large distributed graphs. In: Proceedings of ICDE. Demonstration. (2012)
31. Agarwal, V., Petrini, F., Pasetto, D., Bader, D.A.: Scalable graph exploration on multicore processors. In: Proceedings of ACM/IEEE Supercomputing, Washington, DC, USA, IEEE Computer Society (2010) 1–11
32. Pearce, R., Gokhale, M., Amato, N.M.: Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In: Proceedings of ACM/IEEE Supercomputing, Washington, DC, USA, IEEE Computer Society (2010) 1–11
33. Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., Chen, E.: Kineograph: taking the pulse of a fast-changing and connected world. In: Proceedings of EuroSys, New York, NY, USA, ACM (2012) 85–98
34. Gutiérrez, A., Pucheral, P., Steffen, H., Thévenin, J.M.: Database graph views: A practical model to manage persistent graphs. In: Proceedings of the 20th International Conference on Very Large Data Bases. VLDB (1994)