hardware-assisted approach by reading/resetting the *young* and *dirty* flags of a page. Compression is applied on cold data to further improve the performance of OLAP workload and reduce memory consumption [135].

Snapshot is not only used for OLAP queries, but also for long-running transactions [238], as these long-running transactions will block other short good-natured transactions in the serial execution model. In HyPer, these ill-natured transactions are identified and tentatively executed on a child process with a consistent snapshot, and the changes made by these transactions are effected by issuing a deterministic "apply transaction", back to the main database process. The *apply transaction* validates the execution of the tentative transaction, by checking that all reads performed on the snapshot are identical to what would have been read on the main database if view serializability is required, or by checking the writes on the snapshot are disjoint from the writes by all transactions on the main database after the snapshot was created if the snapshot isolation is required. If the validation succeeds, it applies the writes to the main database state. Otherwise an abort is reported to the client.

Register-conscious compilation. To process a query, HyPer translates it into compact and efficient machine code using the LLVM compiler framework [112], [167], rather than using the classical iterator-based query processing model. The HyPer JIT compilation model is designed to avoid function calls by extending recursive function calls into a code fragment loop, thus resulting in better code locality and data locality (i.e., temporal locality for CPU registers), because each code fragment performs all actions on a tuple within one execution pipeline during which the tuple is kept in the registers, before materializing the result into the memory for the next pipeline.

As an optimized high-level language compiler (e.g., C++) is slow, HyPer uses the LLVM compiler framework to generate portable assembler code for an SQL query. In particular, when processing an SQL query, it is first processed as per normal, i.e., the query is parsed, translated and optimized into an algebraic logical plan. However, the algebraic logical plan is not translated into an executable physical plan as in the conventional scheme, but instead compiled into an imperative program (i.e., LLVM assembler code) which can then be executed directly using the JIT compiler provided by LLVM. Nevertheless, the complex part of query processing (e.g., complex data structure management, sorting) is still written in C++, which is pre-compiled. As the LLVM code can directly call the native C++ method without additional wrapper, C++ and LLVM interact with each other without performance penalty [112]. However, there is a trade-off between defining functions, and inlining code in one compact code fragment, in terms of code cleanness, the size of the executable file, efficiency, etc.

ART Indexing. HyPer uses an adaptive radix tree [87] for efficient indexing. The property of the radix tree guarantees that the keys are ordered bit-wise lexicographically, making it possible for range scan, prefix lookup, etc. Larger span of radix tree can decrease the tree height linearly, thus speeding up the search process, but increase the space consumption exponentially. ART achieves both space and time efficiency by adaptively using different inner node sizes with the same, relatively large span, but different fan-out.

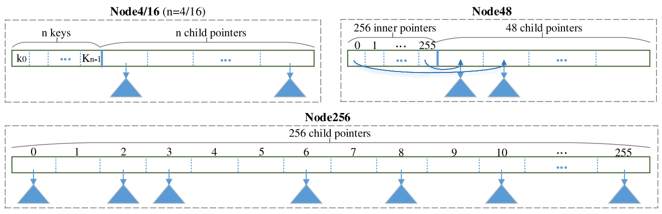

Fig. 5. ART inner node structures.

Specifically, there are four types of inner nodes with a span of 8 bits but different capacities: Node4, Node16, Node48 and Node256, which are named according to their maximum capacity of storing child node pointers. In particular, Node4/Node16 can store up to 4/16 child pointers and uses an array of length 4/16 for sorted keys and another array of the same length for child pointers. Node48 uses a 256-element array to directly index key bits to the pointer array with capacity of 48, while Node256 is simply an array of 256 pointers as normal radix tree node, which is used to store between 49 to 256 entries. Fig. 5 illustrates the structures of Node4, Node16, Node48 and Node256. Lazy expansion and path compression techniques are adopted to further reduce the memory consumption.

3.1.4 SAP HANA

SAP HANA [77], [239], [240] is a distributed in-memory database featured for the integration of OLTP and OLAP [41], and the unification of structured (i.e., relational table) [74], semi-structured (i.e., graph) [241] and unstructured data (i.e., text) processing. All the data is kept in memory as long as there is enough space available, otherwise entire data objects (e.g., tables or partitions) are unloaded from memory and reloaded into memory when they are needed again. HANA has the following features:

- It supports both row- and column-oriented stores for relational data, in order to optimize different query workloads. Furthermore, it exploits columnar data layout for both efficient OLAP and OLTP by adding two levels of delta data structures to alleviate the inefficiency of insertion and deletion operations in columnar data structures [74].
- It provides rich data analytics functionality by offering multiple query language interfaces (e.g., standard SQL, SQLScript, MDX, WIPE, FOX and R), which makes it easy to push down more application semantics into the data management layer, thus avoiding heavy data transfer cost.
- It supports temporal queries based on the *Timeline Index* [242] naturally as data is versioned in HANA.
- It provides snapshot isolation based on multi-version concurrency control, transaction semantics based on optimized two-phase commit protocol (2PC) [243], and fault-tolerance by logging and periodic checkpointing into GPFS file system [148].

We will elaborate only on the first three features, as the other feature is a fairly common technique used in the literature.

Relational stores. SAP HANA supports both row- and column-oriented physical representations of relational tables. Row store is beneficial for heavy updates and inserts, as

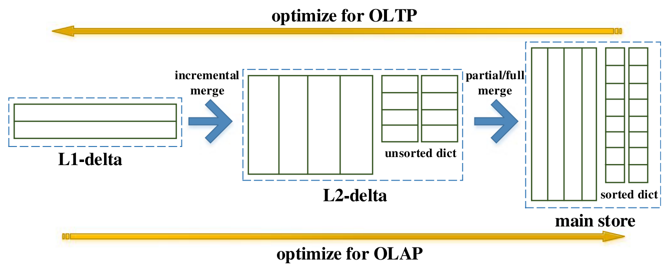

Fig. 6. HANA hybrid store.

well as point queries that are common in OLTP, while column store is ideal for OLAP applications as they usually access all values of a column together, and few columns at a time. Another benefit for column-oriented representation is that it can utilize compression techniques more effectively and efficiently. In HANA, a table/partition can be configured to be either in the row store or in the column store, and it can also be re-structured from one store to the other. HANA also provides a storage advisor [244] to recommend the optimal representation based on data and query characteristics by taking both query cost and compression rate into consideration.

As a table/partition only exists in either a row store or a column store, and both have their own weaknesses, HANA designs a three-level column-oriented unified table structure, consisting of L1-delta, L2-delta and main store, which is illustrated in Fig. 6, to provide efficient support for both OLTP and OLAP workloads, which shows that column store can be deployed efficiently for OLTP as well [41], [74]. In general, a tuple is first stored in L1-delta in row format, then propagated to L2-delta in column format and finally merged with the main store with heavier compression. The whole process of the three stages is called a lifecycle of a tuple in HANA term.

Rich data analytics support. HANA supports various programming interfaces for data analytics (i.e., OLAP), including standard SQL for generic data management functionality, and more specialized languages such as SQL script, MDX, FOX, WIPE [74], [240] and R [245]. While SQL queries are executed in the same manner as in a traditional database, other specialized queries have to be transformed. These queries are first parsed into an intermediate abstract data flow model called “calculation graph model”, where source nodes represent persistent or intermediate tables and inner nodes reflect logical operators performed by these queries, and then transformed into execution plans similar to that of an SQL query. Unlike other systems, HANA supports R scripting as part of the system to enable better optimization of ad-hoc data analytics jobs. Specifically, R scripts can be embedded into a custom operator in the calculation graph [245]. When an R operator is to be executed, a separate R runtime is invoked using the Rserve package [246]. As the column format of HANA column-oriented table is similar to R’s vector-oriented dataframe, there is little overhead in the transformation from table to dataframe. Data transfer is achieved via shared memory, which is an efficient inter-process communication (IPC) mechanism. With the help of *RICE* package [245], it only needs to copy once to make the data available for the R process, i.e., it just copies the data from the database to the shared memory section, and the R runtime can access the data from the shared memory section directly.

Temporal query. HANA supports temporal queries, such as temporal aggregation, time travel and temporal join, based on a unified index structure called the Timeline Index [88], [242], [247]. For every logical table, HANA keeps the *current* version of the table in a *Current Table* and the whole history of previous versions in a *Temporal Table*, accompanied with a *Timeline Index* to facilitate temporal queries. Every tuple of the *Temporal Table* carries a valid interval, from its *commit time* to its *last valid time*, at which some transaction invalidates that value. Transaction Time in HANA is represented by discrete, monotonically increasing *versions*. Basically, the *Timeline Index* maps each *version* to all the write events (i.e., records in the *Temporal Table*) that committed before or at that version. A *Timeline Index* consists of an *Event List* and a *Version Map*, where the *Event List* keeps track of every *invalidation* or *validation* event, and the *Version Map* keeps track of the sequence of events that can be seen by each version of the database. Consequently due to the fact that all visible rows of the Temporal Table at every point in time are tracked, temporal queries can be implemented by scanning *Event List* and *Version Map* concurrently.

To reduce the full scan cost for constructing a temporal view, HANA augments the difference-based *Timeline Index* with a number of complete view representations, called checkpoints, at a specific time in the history. In particular, a checkpoint is a bit vector with length equal to the number of rows in the *Temporal Table*, which represents the visible rows of the *Temporal Table* at a certain time point (i.e., a certain version). With the help of checkpoints, a temporal view at a certain time can be obtained by scanning from the latest checkpoint before that time, rather than scanning from the start of the *Event List* each time.

3.2 In-Memory NoSQL Databases

NoSQL is short for Not Only SQL, and a NoSQL database provides a different mechanism from a relational database for data storage and retrieval. Data in NoSQL databases is usually structured as a tree, graph or key-value rather than a tabular relation, and the query language is usually not SQL as well. NoSQL database is motivated by its simplicity, horizontal scaling and finer control over availability, and it usually compromises consistency in favor of availability and partition tolerance [25], [248].

With the trend of “Memory is the new disk”, in-memory NoSQL databases are flourishing in recent years. There are key-value stores such as Redis [66], RAMCloud [2], MemepiC [60], [138], Masstree [249], MICA [64], Mercury [250], Citrusleaf/Aerospike [34], Kyoto/Tokyo Cabinet [251], Pilaf [252], document stores such as MongoDB [65], Couchbase [253], graph databases such as Trinity [46], Bitsy [254], RDF databases such as OWLIM [255], WhiteDB [50], etc. There are some systems that are partially in-memory, such as MongoDB [65], MonetDB [256], MDB [257], as they use memory-mapped files to store data such that the data can be accessed as if it was in the memory.

In this section, we will introduce some representative in-memory NoSQL databases, including MemepiC [60], [138], MongoDB [65], RAMCloud [2], [75], [126], [258], [259], Redis [66] and some graph databases.

3.2.1 MemepiC

MemepiC [60] is the in-memory version of epiC [23], an extensible and scalable system based on *Actor Concurrent programming model* [260], which has been designed for processing Big Data. It not only provides low latency storage service as a distributed key-value store, but also integrates in-memory data analytics functionality to support online analytics. With an efficient data eviction and fetching mechanism, MemepiC has been designed to maintain data that is much larger than the available memory, without severe performance degradation. We shall elaborate MemepiC in three aspects: system calls reduction, integration of storage service and analytics operations, and virtual memory management.

Less-system-call design. The conventional database design that relies on system calls for communication with hardware or synchronization is no longer suitable for achieving good performance demanded by in-memory systems, as the overhead incurred by system calls is detrimental to the overall performance. Thus, MemepiC subscribes to the less-system-call design principle, and attempts to reduce as much as possible on the use of system calls in the storage access (via memory-mapped file instead), network communication (via RDMA or library-based networking), synchronization (via transactional memory or atomic primitives) and fault-tolerance (via remote logging) [60].

Integration of storage service and analytics operations. In order to meet the requirement of online data analytics, MemepiC also integrates data analytics functionality, to allow analyzing data where it is stored [60]. With the integration of data storage and analytics, it significantly eliminates the data movement cost, which typically dominates in conventional data analytics scenarios, where data is first fetched from the database layer to the application layer, only after which it can be analyzed [172]. The synchronization between data analytics and storage service is achieved based on atomic primitives and *fork*-based virtual snapshot.

User-space virtual memory management (UVMM). The problem of relatively smaller size of main memory is alleviated in MemepiC via an efficient user-space virtual memory management mechanism, by allowing data to be freely evicted to disks when the total data size exceeds the memory size, based on a configurable paging strategy [138]. The adaptability of data storage enables a smooth transition from disk-based to memory-based databases, by utilizing a hybrid of storages. It takes advantage of not only semantics-aware eviction strategy but also hardware-assisted I/O and CPU efficiency, exhibiting a great potential as a more general approach of “Anti-Caching” [138]. In particular, it adopts the following strategies.

- A hybrid of access tracking strategies, including user-supported tuple-level access logging, MMU-assisted page-level access tracking, virtual memory area (VMA)-protection-based method and *malloc*-injection, which achieves light-weight and semantics-aware access tracking.
- Customized WSCLOCK paging strategy based on fine-grained access traces collected by above-mentioned access tracking methods, and other alternative strategies including LRU, aging-based LRU and FIFO, which enables a more accurate and flexible online eviction strategy.
- VMA-protection-based book-keeping method, which incurs less memory overhead for book-keeping the location of data, and tracking the data access in one go.
- Larger data swapping unit with a fast compression technique (i.e., LZ4 [261]) and kernel-supported asynchronous I/O, which can take advantage of the kernel I/O scheduler and block I/O device, and reduce the I/O traffic significantly.

3.2.2 MongoDB

MongoDB [65] is a document-oriented NoSQL database, with few restrictions on the schema of a document (i.e., BSON-style). Specifically, a MongoDB hosts a number of databases, each of which holds a set of collections of documents. MongoDB provides atomicity at the document-level, and indexing and data analytics can only be conducted within a single collection. Thus “cross-collection” queries (such as join in traditional databases) are not supported. It uses primary/secondary replication mechanism to guarantee high availability, and sharding to achieve scalability. In a sense, MongoDB can also act as a cache for documents (e.g., HTML files) since it provides data expiration mechanism by setting TTL (Time-to-Live) for documents.

We will discuss two aspects of MongoDB in detail in the following sections, i.e., the storage and data analytics functionality.

Memory-mapped file. MongoDB utilizes memory-mapped files for managing and interacting with all its data. It can act as a fully in-memory database if the total data can fit into the memory. Otherwise it depends on the virtual-memory manager (VMM) which will decide when and which page to page in or page out. Memory-mapped file offers a way to access the files on disk in the same way we access the dynamic memory—through pointers. Thus we can get the data on disk directly by just providing its pointer (i.e., virtual address), which is achieved by the VMM that has been optimized to make the paging process as fast as possible. It is typically faster to access memory-mapped files than direct file operations because it does not need a system call for normal access operations and it does not require memory copy from kernel space to user space in most operating systems. On the other hand, the VMM is not able to adapt to MongoDB’s own specific memory access patterns, especially when multiple tenants reside in the same machine. A more intelligent ad-hoc scheme would be able to manage the memory more effectively by taking specific usage scenarios into consideration.

Data analytics. MongoDB supports two types of data analytics operations: aggregation (i.e., aggregation pipeline and single purpose aggregation operations in MongoDB term) and MapReduce function which should be written in JavaScript language. Data analytics on a sharded cluster that needs central assembly is conducted in two steps:

- The query router divides the job into a set of tasks and distributes the tasks to the appropriate sharded instances, which will return the partial results back to the query router after finishing the dedicated computations.
- The query router will then assemble the partial results and return the final result to the client.

3.2.3 RAMCloud

RAMCloud [2], [75], [126], [258], [259] is a distributed in-memory key-value store, featured for low latency, high availability and high memory utilization. In particular, it can achieve tens of microseconds latency by taking advantage of low-latency networks (e.g., Infiniband and Myrinet), and provide "continuous availability" by harnessing large scale to recover in 1-2 seconds from system failure. In addition, it adopts a log-structured data organization with a two-level cleaning policy to structure the data both in memory and on disks. This results in high memory utilization and a single unified data management strategy. The architecture of RAMCloud consists of a coordinator who maintains the metadata in the cluster such as cluster membership, data distribution, and a number of storage servers, each of which contains two components, a master module which manages the in-memory data and handles read/write requests from clients, and a backup module which uses local disks or flash memory to backup replicas of data owned by other servers.

Data organization. Key-value objects in RAMCloud are grouped into a set of tables, each of which is individually range-partitioned into a number of tablets based on the hash-codes of keys. RAMCloud relies on the uniformity of hash function to distribute objects in a table evenly in proportion to the amount of hash space (i.e., the range) a storage server covers. A storage server uses a single log to store the data, and a hash table for indexing. Data is accessed via the hash table, which directs the access to the current version of objects.

RAMCloud adopts a log-structured approach of memory management rather than traditional memory allocation mechanisms (e.g., C library's *malloc*), allowing 80-90 percent memory utilization by eliminating memory fragmentation. In particular, a log is divided into a set of *segments*. As the log structure is append-only, objects are not allowed to be deleted or updated in place. Thus a periodic clean job should be scheduled to clean up the deleted/stale objects to reclaim free space. RAMCloud designs an efficient two-level cleaning policy.

- It schedules a *segment compaction* job to clean the log segment in memory first whenever the free memory is less than 10 percent, by copying its live data into a smaller *segment* and freeing the original *segment*.
- When the data on disk is larger than that in memory by a threshold, a *combined cleaning* job starts, cleaning both the log in memory and on disk together.

A two-level cleaning policy can achieve a high memory utilization by cleaning the in-memory log more frequently, and meanwhile reduce disk bandwidth requirement by trying to lower the disk utilization (i.e., increase the percentage of deleted/stale data) since this can avoid copying a large percentage of live data on disk during cleaning.

Fast crash recovery. One big challenge for in-memory storage is fault-tolerance, as the data is resident in the volatile DRAM. RAMCloud uses replication to guarantee durability by replicating data in remote disks, and harnesses the large scale of resources (e.g., CPU, disk bandwidth) to speed up recovery process [126], [259]. Specifically, when receiving an update request from a client, the master server appends the new object to the in-memory log, and then forwards the object to R (usually $R = 3$) remote backup servers, which buffer the object in memory first and flush the buffer onto disk in a batch (i.e., in unit of *segment*). The backup servers respond as soon as the object has been copied into the buffer, thus the response time is dominated by the network latency rather than the disk I/O.

To make recovery faster, replicas of the data are scattered across all the backup servers in the cluster in unit of *segment*, thus making more backup servers collaborate for the recovery process. Each master server decides independently where to place a segment replica using a combination of randomization and refinement, which not only eliminates pathological behaviors but also achieves a nearly optimal solution. Furthermore, after a server fails, in addition to all the related backup servers, multiple master servers are involved to share the recovery job (i.e., re-constructing the in-memory log and hash table), and take responsibility for ensuring an even partitioning of the recovered data. The assignment of recovery job is determined by a *will* made by the master before it crashes. The *will* is computed based on *tablet profiles*, each of which maintains a histogram to track the distribution of resource usage (e.g., the number of records and space consumption) within a single table/tablet. The *will* aims to balance the partitions of a recovery job such that they require roughly equal time to recover.

The random replication strategy produces almost uniform allocation of replicas and takes advantage of the large scale, thus preventing data loss and minimizing recovery time. However, this strategy may result in data loss under simultaneous node failures [125]. Although the amount of lost data may be small due to the high dispersability of segment replicas, it is possible that all replicas of certain part of the data may become unavailable. Hence RAMCloud also supports another replication mode based on *Copyset* [125], [258], [259], to reduce the probability of data loss after large, coordinated failures such as power loss. Copyset trades off the amount of lost data for the reduction in the frequency of data loss, by constraining the set of backup servers where all the segments in a master server can be replicated to. However, this can lead to longer recovery time as there are fewer backup servers for reading the replicas from disks. The trade-off can be controlled by the scatter width, which is the number of backup servers that each server's data are allowed to be replicated to. For example, if the scatter width equals the number of all the other servers (except the server that wants to replicate) in the cluster, Copyset then turns to random replication.

3.2.4 Redis

Redis [66] is an in-memory key-value store implemented in C with support for a set of complex data structures, including hash, list, set, sorted set, and some advanced functions such as publish/subscribe messaging, scripting and transactions. It also embeds two persistence mechanisms—snapshotting and append-only logging. Snapshotting will back up all the current data in memory onto disk periodically, which facilitates recovery process, while append-only logging will log every update operation, which guarantees more availability. Redis is single-threaded, but it processes requests asynchronously by utilizing an event notification strategy to overlap the network I/O communication and data storage/retrieval computation.

Redis also maintains a hash-table to structure all the key-value objects, but it uses naive memory allocation (e.g., malloc/free), rather than slab-based memory allocation strategy (i.e., Memcached's), thus making it not very suitable as an LRU cache, because it may incur heavy memory fragmentation. This problem is partially alleviated by adopting the *jemalloc* [262] memory allocator in the later versions.

Scripting. Redis features the server-side scripting functionality (i.e., Lua scripting), which allows applications to perform user-defined functions inside the server, thus avoiding multiple round-trips for a sequence of dependent operations. However, there is an inevitable costly overhead in the communication between the scripting engine and the main storage component. Moreover, a long-running script can degenerate the overall performance of the server as Redis is single-threaded and the long-running script can block all other requests.

Distributed Redis. The first version of distributed Redis is implemented via data sharding on the client-side. Recently, the Redis group introduces a new version of distributed Redis called Redis Cluster, which is an autonomous distributed data store with support for automatic data sharding, master-slave fault-tolerance and online cluster re-organization (e.g., adding/deleting a node, re-sharding the data). Redis Cluster is fully distributed, without a centralized master to monitor the cluster and maintain the metadata. Basically, a Redis Cluster consists of a set of Redis servers, each of which is aware of the others. That is, each Redis server keeps all the metadata information (e.g., partitioning configuration, aliveness status of other nodes) and uses gossip protocol to propagate updates.

Redis Cluster uses a hash slot partition strategy to assign a subset of the total hash slots to each server node. Thus each node is responsible for the key-value objects whose hash code is within its assigned slot subset. A client is free to send requests to any server node, but it will get *redirection* response containing the address of an appropriate server when that particular node cannot answer the request locally. In this case, a single request needs two round-trips. This can be avoided if the client can cache the map between hash slots and server nodes. The current version of Redis Cluster requires manual re-sharding of the data and allocating of slots to a newly-added node. The availability is guaranteed by accompanying a master Redis server with several slave servers which replicate all the data of the master, and it uses asynchronous replication in order to gain good performance, which, however, may introduce inconsistency among primary copy and replicas.

3.2.5 In-Memory Graph Databases

Bitsy. Bitsy [254] is an embeddable in-memory graph database that implements the *Blueprints API*, with ACID guarantees on transactions based on the optimistic concurrency model. Bitsy maintains a copy of the entire graph in memory, but logs every change to the disk during a commit operation, thus enabling recovery from failures. Bitsy is designed to work in multi-threaded OLTP environments. Specifically, it uses multi-level dual-buffer/log to improve the write transaction performance and facilitate log cleaning, and lock-free reading with sequential locks to ameliorate the read performance. Basically, it has three main design principles:

- *No seek*. Bitsy appends all changes to an unordered transaction log, and depends on re-organization process to clean the obsolete vertices and edges.
- *No socket*. Bitsy acts as an embedded database, which is to be integrated into an application (java-based). Thus the application can access the data directly, without the need to transfer through socket-based interface which results in a lot of system calls and serialization/de-serialization overhead.
- *No SQL*. Bitsy implements the Blueprints API, which is oriented for the property graph model, instead of the relational model with SQL.

Trinity. Trinity is an in-memory distributed graph database and computation platform for graph analytics [46], [263], whose graph model is built based on an in-memory key-value store. Specifically, each graph node corresponds to a Trinity cell, which is an (*id*, blob) pair, where *id* represents a node in the graph, and the blob is the adjacent list of the node in serialized binary format, instead of runtime object format, in order to minimize memory overhead and facilitate checkpointing process. However this introduces serialization/deserialization overhead in the analytics computation. A large cell, i.e., a graph node with a large number of neighbors, is represented as a set of small cells, each of which contains only the local edge information, and a central cell that contains cell *id*s for the dispersed cells. Besides, the edge can be tagged with a label (e.g., a predicate) such that it can be extended to an RDF store [263], with both local predicate indexing and global predicate indexing support. In local predicate indexing, the adjacency list for each node is sorted first by predicates and then by neighboring nodes *id*, such as SPO[11] or OPS index in traditional RDF store, while the global predicate index enables the locating of cells with a specific predicate as traditional PSO or POS index.

3.3 In-Memory Cache Systems

Cache plays an important role in enhancing system performance, especially in web applications. Facebook, Twitter, Wikipedia, LiveJournal, et al. are all taking advantage of cache extensively to provide good service. Cache can provide two optimizations for applications: optimization for disk I/O by allowing to access data from memory, and optimization for CPU workload by keeping results without the need for re-computation. Many cache systems have been developed for

11. S stands for subject, P for predicate, and O for object.

various objectives. There are general cache systems such as Memcached [61] and BigTable Cache [248], systems targeting speeding up analytics jobs such as PACMan [264] and GridGain [51], and more purpose specific systems that have been designed for supporting specific frameworks such as NCache [265] for .NET and Velocity/AppFabric [266] for Windows servers, systems supporting strict transactional semantics such as TxCache [267], and network caching such as HashCache [268].

Nonetheless, cache systems were mainly designed for web applications, as Web 2.0 increases both the complexity of computation and strictness of service-level agreement (SLA). Full-page caching [269], [270], [271] was adopted in the early days, while it becomes appealing to use fine-grained object-level data caching [61], [272] for flexibility. In this section, we will introduce some representative cache systems/libraries and their main techniques in terms of in-memory data management.

3.3.1 Memcached

Memcached [61] is a light-weight in-memory key-value object caching system with strict LRU eviction. Its distributed version is achieved via the client-side library. Thus, it is the client libraries that manage the data partitioning (usually hash-based partitioning) and request routing. Memcached has different versions of client libraries for various languages such as C/C++, PHP, Java, Python, etc. In addition, it provides two main protocols, namely *text protocol* and *binary protocol*, and supports both UDP and TCP connections.

Data organization. Memcached uses a big hash-table to index all the key-value objects, where the key is a text string and the value is an opaque byte block. In particular, the memory space is broken up into slabs of 1 MB, each of which is assigned to a slab class. And slabs do not get re-assigned to another class. The slab is further cut into chunks of a specific size. Each slab class has its own chunk size specification and eviction mechanism (i.e., LRU). Key-value objects are stored in the corresponding slabs based on their sizes. The slab-based memory allocation is illustrated in Fig. 7, where the grow factor indicates the chunk size difference ratio between adjacent slab classes. The slab design helps prevent memory fragmentation and optimize memory usage, but also causes slab calcification problems. For example, it may incur unnecessary evictions in scenarios where Memcached tries to insert a 500 KB object when it runs out of 512 KB slabs but has lots of 2 MB slabs. In this case, a 512 KB object will be evicted although there is still a lot of free space. This problem is alleviated in the optimized versions of Facebook and Twitter [273], [274].

Concurrency. Memcached uses *libevent* library to achieve asynchronous request processing. In addition, Memcached is a multi-threaded program, with fine-grained *pthread mutex* lock mechanism. A *static* item lock hash-table is used to control the concurrency of memory accesses. The size of the lock hash-table is determined based on the configured number of threads. And there is a trade-off between the memory usage for the lock hash-table and the degree of parallelism. Even though Memcached provides such a fine-grained locking mechanism, most of operations such as index lookup/update and cache eviction/update still need global locks [63], which prevents current Memcached from scaling up on multi-core CPUs [275].

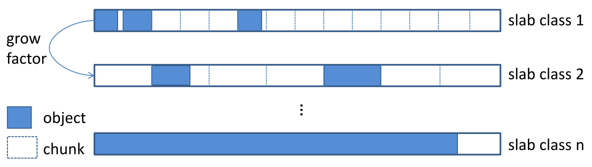

Fig. 7. Slab-based allocation.

Memcached in Production—Facebook's Memcache [273] *and* Twitter's *Twemcache [274]*. Facebook scales Memcached at three different deployment levels (i.e., cluster, region and across regions) from the engineering point of view, by focusing on its specific workload (i.e., read-heavy) and trading off among performance, availability and consistency [273]. Memcache improves the performance of Memcached by designing fine-grained locking mechanism, adaptive slab allocator and a hybrid of lazy and proactive eviction schemes. Besides, Memcache focuses more on the deployment-level optimization. In order to reduce the latency of requests, it adopts parallel requests/batching, uses connection-less UDP for *get* requests and incorporates flow-control mechanisms to limit incast congestion. It also utilizes techniques such as leases [276] and stale reads [277] to achieve high hit rate, and provisioned pools to balance load and handle failures.

Twitter adopts similar optimizations on its distributed version of Memcached, called Twemcache. It alleviates Memcached's slab allocation problem (i.e., slab calcification problem) by random eviction of a whole slab and re-assignment of a desired slab class, when there is not enough space. It also enables a lock-less stat collection via the updater-aggregator model, which is also adopted by Facebook's Memcache. In addition, Twitter also provides a proxy for the Memcached protocol, which can be used to reduce the TCP connections in a huge deployment of Memcached servers.

3.3.2 MemC3

MemC3 [63] optimizes Memcached in terms of both performance and memory efficiency by using optimistic concurrent *cuckoo* hashing and LRU-approximating eviction algorithm based upon CLOCK [279], with the assumption that small and read-only requests dominate in real-world workloads. MemC3 mostly facilitates read-intensive workloads, as the write operations are still serialized in MemC3 and *cuckoo* hashing favors read over write operation. In addition, applications involving a large number of small objects should benefit more from MemC3 in memory efficiency because MemC3 eliminates a lot of pointer overhead embedded in the key-value object. CLOCK-based eviction algorithm takes less memory than list-based strict LRU, and makes it possible to achieve high concurrency as it needs no global synchronization to update LRU.

Optimistic Concurrent Cuckoo *Hashing*. The basic idea of *cuckoo* hashing [278] is to use two hash functions to provide each key two possible buckets in the hash table. When a new key is inserted, it is inserted into one of its two possible buckets. If both buckets are occupied, it will randomly displace the key that already resides in one of these two buckets. The displaced key is then inserted into its alternative bucket, which may further trigger a displacement, until a vacant slot is found or until a maximum number of displacements is reached (at this point, the hash table is rebuilt using new hash functions). This sequence of displacements forms a *cuckoo* displacement path. The collision resolution strategy of *cuckoo* hashing can achieve a high load factor. In addition, it eliminates the pointer field embedded in each key-value object in the chaining-based hashing used by Memcached, which further ameliorates the memory efficiency of MemC3, especially for small objects.

MemC3 optimizes the conventional *cuckoo* hashing by allowing each bucket with four tagged slots (i.e., four-way set-associative), and separating the discovery of a valid *cuckoo* displacement path from the execution of the path for high concurrency. The tag in the slot is used to filter the unmatched requests and help to calculate the alternative bucket in the displacement process. This is done without the need for the access to the exact key (thus no extra pointer de-reference), which makes both look-up and insert operations cache-friendly. By first searching for the *cuckoo* displacement path and then moving keys that need to be displaced backwards along the *cuckoo* displacement path, it facilitates fine-grained optimistic locking mechanism. MemC3 uses lock striping techniques to balance the granularity of locking, and optimistic locking to achieve multiple-reader/single-writer concurrency.

3.3.3 TxCache

TxCache [267] is a snapshot-based transactional cache used to manage the cached results of queries to a transactional database. TxCache ensures that transactions see only consistent snapshots from both the cache and the database, and it also provides a simple programming model where applications simply designate functions/queries as cacheable and the TxCache library handles the caching/invalidating of results.

TxCache uses versioning to guarantee consistency. In particular, each object in the cache and the database is tagged with a version, described by its validity interval, which is a range of timestamps at which the object is valid. A transaction can have a staleness condition to indicate that the transaction can tolerate a consistent snapshot within the past staleness seconds. Thus only records that overlap with the transaction's tolerance range (i.e., the range between its timestamp minus staleness and its timestamp) should be considered in the transaction execution. To increase the cache hit rate, the timestamp of a transaction is chosen lazily by maintaining a set of satisfying timestamps and revising it while querying the cache. In this way, the probability of getting more requested records from the cache increases. Moreover, it still keeps the multi-version consistency at the same time. The cached results are automatically invalidated whenever their dependent records are updated. This is achieved by associating each object in the cache with an invalidation tag, which describes which parts of the database it depends on. When some records in the database are modified, the database identifies the set of invalidation tags affected and passes these tags to the cache nodes.

4 In-Memory Data Processing Systems

In-memory data processing/analytics is becoming more and more important in the Big Data era as it is necessary to analyze a large amount of data in a small amount of time. In general, there are two types of in-memory processing systems: data analytics systems which focus on batch processing such as Spark [55], Piccolo [59], SINGA [280], Pregel [281], GraphLab [47], Mammoth [56], Phoenix [57], GridGain [51], and real-time data processing systems (i.e., stream processing) such as Storm [53], Yahoo! S4 [52], Spark Streaming [54], MapReduce Online [282]. In this section, we will review both types of in-memory data processing systems, but mainly focus on those designed for supporting data analytics.

4.1 In-Memory Big Data Analytics Systems

4.1.1 Main Memory MapReduce (M3R)

M3R [58] is a main memory implementation of MapReduce framework. It is designed for interactive analytics with terabytes of data which can be held in the memory of a small cluster of nodes with high mean time to failure. It provides a backward compatible interfaces with conventional MapReduce [283], and significantly better performance. However, it does not guarantee resilience because it caches the results in memory after map/reduce phase instead of flushing into the local disk or HDFS, making M3R not suitable for long-running jobs. Specifically, M3R optimizes the conventional MapReduce design in two aspects as follows:

- It caches the input/output data in an in-memory key-value store, such that the subsequent jobs can obtain the data directly from the cache and the materialization of output results is eliminated. Basically, the key-value store uses a path as a key, and maps the path to a metadata location where it contains the locations for the data blocks.
- It guarantees partition stability to achieve locality by specifying a partitioner to control how keys are mapped to partitions amongst reducers, thus allowing an iterative job to re-use the cached data.

4.1.2 Piccolo

Piccolo [59] is an in-memory data-centric programming framework for running data analytics computation across multiple nodes with support for data locality specification and data-oriented accumulation. Basically, the analytics program consists of a control function which is executed on the master, and a kernel function which is launched as multiple instances concurrently executing on many worker nodes and sharing distributed mutable key-value tables, which can be updated on the fine-grained key-value object level. Specifically, Piccolo supports the following functionalities:

- A user-defined accumulation function (e.g., max, summation) can be associated with each table, and Piccolo executes the accumulation function during runtime to combine concurrent updates on the same key.
- To achieve data locality during the distributed computation, users are allowed to define a partition

function for a table and co-locate a kernel execution with some table partition or co-locate partitions from different tables.

- Piccolo handles machine failures via a global user-assisted checkpoint/restore mechanism, by explicitly specifying when and what to checkpoint in the control function.
- Load-balance during computation is optimized via work stealing, i.e., a worker that has finished all its assigned tasks is instructed to steal a not-yet-started task from the worker with the most remaining tasks.

4.1.3 Spark/RDD

Spark system [55], [284] presents a data abstraction for big data analytics, called resilient distributed dataset (RDD), which is a coarse-grained deterministic immutable data structure with lineage-based fault-tolerance [285], [286]. On top of Spark, Spark SQL, Spark Streaming, MLlib and GraphX are built for SQL-based manipulation, stream processing, machine learning and graph processing, respectively. It has two main features:

- It uses an elastic persistence model to provide the flexibility to persist the dataset in memory, on disks or both. By persisting the dataset in memory, it favors applications that need to read the dataset multiple times (e.g., iterative algorithms), and enables interactive queries.
- It incorporates a light-weight fault-tolerance mechanism (i.e., lineage), without the need for checkpointing. The lineage of an RDD contains sufficient information such that it can be re-computed based on its lineage and dependent RDDs, which are the input data files in HDFS in the worst case. This idea is also adopted by *Tachyon* [287], which is a distributed file system enabling reliable file sharing via memory.

The ability of persisting data in memory in a fault-tolerance manner makes RDD suitable for many data analytics applications, especially those iterative jobs, since it removes the heavy cost of shuffling data onto disks at every stage as Hadoop does. We elaborate on the following two aspects of RDD: data model and job scheduling.

Data model. RDD provides an abstraction for a read-only distributed dataset. Data modification is achieved by coarse-grained RDD transformations that apply the same operation to all the data items in the RDD, thus generating a new RDD. This abstraction offers opportunities for high consistency and a light-weight fault-tolerance scheme. Specifically, an RDD logs the transformations it depends on (i.e., its lineage), without data replication or checkpointing for fault-tolerance. When a partition of the RDD is lost, it is re-computed from other RDDs based on its lineage. As RDD is updated by coarse-grained transformations, it usually requires much less space and effort to back up the lineage information than the traditional data replication or checkpointing schemes, at the price of a higher re-computation cost for computation-intensive jobs, when there is a failure. Thus, for RDDs with long lineage graphs involving a large re-computation cost, checkpointing is used, which is more beneficial.

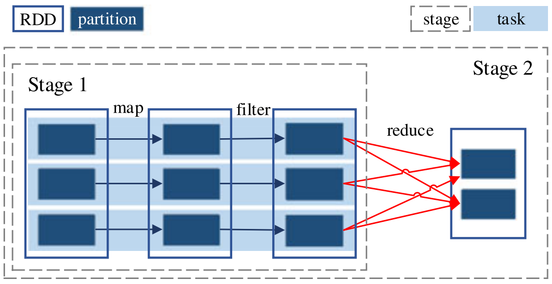

Fig. 8. Spark job scheduler.

The RDD model provides a good caching strategy for "working sets" during computation, but it is not general enough to support traditional data storage functionality for two reasons:

- RDD fault-tolerance scheme is based on the assumption of coarse-grained data manipulation without in-place modification, because it has to guarantee that the program size is much less than the data size. Thus, fine-grained data operations such as updating a single key-value object cannot be supported in this model.
- It assumes that there exists an original dataset persistent on a stable storage, which guarantees the correctness of the fault-tolerance model and the suitability of the block-based organization model. However, in traditional data storage, data is arriving dynamically and the allocation of data cannot be determined beforehand. As a consequence, data objects are dispersed in memory, which results in degraded memory throughput.

Job scheduling. The jobs in Spark are organized into a DAG, which captures job dependencies. RDD uses lazy materialization, i.e., an RDD is not computed unless it is used in an action (e.g., *count()*). When an action is executed on an RDD, the scheduler examines the RDD's lineage to build a DAG of jobs for execution. Spark uses a two-phase job scheduling as illustrated in Fig. 8 [55]:

- It first organizes the jobs into a DAG of stages, each of which may contain a sequence of jobs with only one-to-one dependency on the partition-level. For example, in Fig. 8, Stage 1 consists of two jobs, i.e., map and filter, both of which only have one-to-one dependencies. The boundaries of the stages are the operations with shuffle (e.g., reduce operation in Fig. 8), which have many-to-many dependencies.
- In each stage, a task is formed by a sequence of jobs on a partition, such as the map and filter jobs on the shaded partitions in Fig. 8. Task is the unit of scheduling in the system, which eliminates the materialization of the intermediate states (e.g., the middle RDD of Stage 1 in Fig. 8), and enables a fine-grained scheduling strategy.

4.2 In-Memory Real-Time Processing Systems

4.2.1 Spark Streaming

Spark Streaming [54] is a fault-tolerant stream processing system built based on Spark [55]. It structures a streaming computation as a series of stateless, deterministic batch computations on small time intervals (say 1 s), instead of keeping continuous, stateful operators. Thus it targets applications that tolerate latency of several seconds. Spark Streaming fully utilizes the immutability of RDD and lineage-based fault-tolerance mechanism from Spark, with some extensions and optimizations. Specifically, the incoming stream is divided into a sequence of immutable RDDs based on time intervals, called D-streams, which are the basic units that can be acted on by deterministic transformations, including not only many of the transformations available on normal Spark RDDs (e.g., map, reduce and groupBy), but also windowed computations exclusive for Spark Streaming (e.g., reduceByWindow and countByWindow). RDDs from historical intervals can be automatically merged with the newly-generated RDD as new streams arrive. Stream data is replicated across two worker nodes to guarantee durability of the original data that the lineage-based recovery relies on, and checkpointing is conducted periodically to reduce the recovery time due to long lineage graphs. The determinism and partition-level lineage of D-streams makes it possible to perform parallel recovery after a node fails and mitigate straggler problem by speculative execution.

4.2.2 Yahoo! S4

S4 (Simple Scalable Streaming System) [52] is a fully decentralized, distributed stream processing engine inspired by the MapReduce [283] and Actors model [99]. Basically, computation is performed by processing elements (PEs) which are distributed across the cluster, and messages are transmitted among them in the form of data events, which are routed to corresponding PEs based on their identities. In particular, an event is identified by its type and key, while a PE is defined by its functionality and the events that it intends to consume. The incoming stream data is first transformed as a stream of events, which will then be processed by a series of PEs that are defined by users for specific applications. However, S4 does not provide data fault-tolerance by design, since even though automatic PE failover to standby nodes is supported, the states of the failed PEs and messages are lost during the handoff if there is no user-defined state/message backup function inside the PEs.

5 Qualitative Comparison

In this section, we summarize some representative in-memory data management systems elaborated in this paper in terms of data model, supported workloads, indexes, concurrency control, fault-tolerance, memory overflow control, and query processing strategy in Table 3.

In general, in-memory data management systems can also be classified into three categories based on their functionality such as storage and data analytics, namely storage systems, analytics systems, and full-fledged systems that have both capabilities:

- In-memory storage systems have been designed purely for efficient storage service, such as in-memory relational databases only for OLTP (e.g., H-Store[12] [36], Silo [39], Microsoft Hekaton [37]), NoSQL databases without analytics support (e.g., RAMCloud [75], Masstree [249], MICA [64], Mercury [250], Kyoto/Tokyo Cabinet [251], Bitsy [254]), cache systems (e.g., Memcached [61], MemC3 [63], TxCache [267], HashCache [268]), etc. Storage service focuses more on low latency and high throughput for short-running query jobs, and is equipped with a light-weight framework for online queries. It usually acts as the underlying layer for upper-layer applications (e.g., web server, ERP), where fast response is part of the service level agreement.
- In-memory analytics systems are designed for large scale data processing and analytics, such as in-memory big data analytics systems (e.g., Spark/RDD [55], Piccolo [59], Pregel [281], GraphLab [47], Mammoth [56], Phoenix [57]), and real-time in-memory processing systems (e.g., Storm [53], Yahoo! S4 [52], Spark Streaming [54], MapReduce Online [282]). The main optimization objective of these systems is to minimize the runtime of an analytics job, by achieving high parallelism (e.g., multi-core, distribution, SIMD, and pipelining) and batch processing.
- In-memory full-fledged systems include not only in-memory relational databases with support for both OLTP and OLAP (e.g., HyPer [35], Crescando [227], HYRISE [176]), but also data stores with general purpose query language support (e.g., SAP HANA [77], Redis [66], MemepiC [60], [138], Citrusleaf/Aerospike [34], GridGain [51], MongoDB [65], Couchbase [253], MonetDB [256], Trinity [46]). One major challenge for this category of systems is to make a reasonable tradeoff between two different workloads, by making use of appropriate data structures and organization, resource contention, etc.; concurrency control is also very important as it deals with simultaneous mixed workloads.

6 Research Opportunities

In this section, we briefly discuss the research challenges and opportunities for in-memory data management, in the following optimization aspects, which have been introduced earlier in Table 1:

- Indexing. Existing works on indexing for in-memory databases attempt to optimize both time and space efficiency. Hash-based index is simple and easy to implement, and also offers O(1) access time complexity, while tree-based index supports range query naturally and usually has good space efficiency. Trie-based index has bounded $O(k)$ time complexity, where k is the length of the key. There are also other kinds of indexes such as bitmaps and skip-lists, which are amenable to efficient in-memory and distributed processing. For example, the skip-list, which

12. Based on the H-Store website, it now incorporates a new experimental OLAP engine based on JVM snapshot. Based on its main focus, we put it in the storage category.

TABLE 3
Comparison of In-Memory Data Management Systems

	Systems	Data Model	Workloads	Indexes	Concurrency Control (CC)	Fault Tolerance	Memory Overflow	Query Processing
Relational Databases	**H-Store**	relational (row)	OLTP	hashing, B^+-tree, binary tree	partition, serial execution, light-weight locking, speculative CC	command logging, checkpoint, replica	anti-caching	stored procedure
	Hekaton	relational (row)	OLTP	latch-free hashing, Bw-tree	optimistic MVCC	logging, checkpoint, replica	Project Siberia	complied stored procedure
	HyPer/ ScyPer	relational	OLTP, OLAP	hashing, balanced search tree, ART	virtual snapshot, strict timestamp ordering (STO), partition, serial execution for OLTP	logging, checkpoint, replica	compression	JIT, stored procedure
	SAP HANA	relational, graph, text	OLTP, OLAP	timeline index, CSB^+-tree, inverted index	MVCC, 2PC	logging, checkpoint, standby server, GPFS	table/partition-level swapping, compression	"calculation graph model"
NoSQL Databases	**MemepiC**	key-value	object operations, analytics	hashing, skip-list	atomic primitives, virtual snapshot	logging, replica	user-space VMM	JIT
	MongoDB	document (bson)	object operations, analytics	B-tree	database-level locking	memory-mapped file	N/A	N/A
	RAMCloud	key-value	object operations	hashing	fine-grained locking	logging, replica	N/A	N/A
	Redis	key-value	object operations	hashing	single-threaded	logging, checkpoint	compression	scripting
Graph Databases	**Bitsy**	graph	OLTP	N/A	optimistic concurrency control (version)	logging, backup	N/A	stored procedure
	Trinity	graph	graph operations	N/A	fine-grained spin-lock	replica, Trinity File System (TFS)	N/A	stored procedure
Cache Systems	**Memcached**	key-value	object operations	hashing	fine-grained locking	N/A	N/A	N/A
	MemC3	key-value	object operations	hashing	lock striping, optimistic locking	N/A	N/A	N/A
	TxCache	key-value	OLTP	hashing	MVCC	N/A	N/A	N/A
Big Data Analytics Systems	**M3R**	key-value	analytics	N/A	partition, locking	N/A	N/A	offline
	Piccolo	key-value	analytics	hashing	locking	checkpoint	N/A	offline
	Spark/ RDD	RDD	analytics	N/A	partition, read/write locking	lineage, checkpoint	block-level swapping	offline
Real-time Processing Systems	**Spark Streaming**	RDD	streaming	N/A	partition, read/write locking	lineage, replica, checkpoint	block-level swapping	N/A
	Yahoo! S4	Event	streaming	hashing	message passing	standby server	N/A	N/A

allows fast point- and range-queries of an ordered sequence of elements with O($log\ n$) complexity, is becoming a desirable alternative to B-trees for in-memory databases, since it can be implemented latch-free easily as a result of its layered structures. Indexes for in-memory databases are different from those for disk-based databases, which focus on I/O efficiency rather than memory and cache utilization. It would be very useful to design an index with constant time complexity for point accesses achieved by hash- and trie-based indexes, efficient support for range accesses achieved by tree- and trie-based indexes, and good space efficiency achieved by hash- and tree-based indexes (like ART index [87]). Lock-free or lock-less index structures are essential to achieve high parallelism without latch-related bottleneck, and index-less design or lossy index is also interesting because of its high throughput and low latency of DRAM [41], [64].

- Data layouts. The data layout or data organization is essential to the performance of an in-memory system as a whole. Cache-conscious design such as columnar structure, cache-line alignment, and space utilization optimization such as compression, data de-fragmentation are the main focuses in the in-memory data organization. The idea of continuous data allocation as log structure has been introduced in main memory systems to eliminate the data fragmentation problem and simplify concurrency control [2]. But it may be better to design an application-independent data allocator with common built-in functionality for in-memory systems such as fault-tolerance, and application-assisted data compression and de-fragmentation.
- Parallelism. Three levels of parallelism should be exploited in order to speed up the processing, which have been detailed in Section 1. It is usually beneficial to increase parallelism at the instruction level (e.g., bit-level parallelism, SIMD) provided in modern architecture, which can achieve nearly optimal speedup, free from concurrency issues and other overhead incurred, but with constraints on the maximum parallelism allowed and data structures to operate on. The instruction-level parallelism may yield a good performance boost, and therefore it should be considered in the design of an efficient in-memory data management system, especially in the design of data structures. With the emergence of

many integrated core (MIC) co-processors (e.g., Intel Xeon Phi), it provides a promising alternative for parallelizing computation, with wider SIMD instructions, many lower-frequency in-order cores and hardware contexts [288].

- Concurrency control/transaction management. For in-memory systems, the overhead of concurrency control significantly affects the overall performance, thus making it perfect if there are no concurrency control at all. Hence, it is worth making the serial execution strategy more efficient for cross-partition transactions and more robust to skewed workloads. Lock-less or lock-free concurrency control mechanism is promising in in-memory data management as a heavy-weight lock-based mechanism can greatly offset the performance improved by the in-memory environment. Atomic primitives provided in most mainstream programming languages are efficient alternatives that can be exploited in designing a lock-free concurrency control mechanism. Besides, HTM provides a hardware-assisted approach for efficient concurrency control protocol, especially under the transactional semantics in databases. Hardware-assisted approaches are good choices in the latency-sensitive in-memory environment, as software solutions usually incur heavy overhead that negates the benefits brought by parallelism and fast data access. But we should take care of its unexpected aborts under certain conditions. A mix of these data protection mechanisms (i.e., HTM, lock, timestamp, atomic primitives) should enable a more efficient concurrency control model. Moreover, the protocol should be data-locality sensitive and cache aware, which matter more for modern machines [192].
- Query processing. Query processing is a widely studied research topic even in traditional disk-based databases. However, traditional query processing framework based on Iterator-/Volcano-style model, although flexible, is no longer suitable for in-memory databases because of its poor code/data locality. The high computing power of modern CPU, and easy-to-use compiler infrastructure such as LLVM [167] enable efficient dynamic compiling [112], which can improve the query processing performance significantly as a result of better code and data locality. SIMD or multi-core boosted processing can be utilized to speed up complex database operations such as join and sort, and NUMA architecture will play a bigger role in the future years.
- Fault tolerance. Fault tolerance is a necessity for an in-memory database in order to guarantee durability; however, it is also a major performance bottleneck caused by I/Os. Thus one design philosophy for fault-tolerance is to make it almost invisible to normal operations by minimizing the I/O cost in the critical path as much as possible. Command logging [131] can reduce the data that needs to be logged, while remote logging used by RAMCloud [2] and 2-Safe visible policy of solidDB [40] can reduce the response time by logging the data in remote nodes and replying back as soon as the data is written into the buffer. Fast recovery can provide high availability upon failure, which may be achievable at the price of more and well-organized backuped files (log/checkpoint). The tradeoff between the interference to the normal performance and the recovery efficiency should be further examined [132]. Hardware/OS-assisted approaches are promising, e.g., NVRAM, memory-mapped file, on top of which optimized algorithms and data structures are required to exert its performance potential.
- Data overflow. In general, approaches to the data overflow problem can be classified into three categories: user-space (e.g., H-Store Anti-caching [133], Hekaton Siberia [134]), kernel-space (e.g., OS Swap, MongoDB memory mapped files [65]) and the hybrid (e.g., Efficient OS Paging [136] and UVMM [138]). The semantics-aware user-space approaches can make more effective decision on the paging strategies, while the hardware-conscious and well-developed kernel-space approaches are able to utilize the I/O efficiency brought by the OS during swapping. Potentially, both the semantics-aware paging strategy and hardware-conscious I/O management can be exploited to boost the performance [136], [138].

In addition to the above, hardware solutions are being increasingly exploited for performance gain. In particular, new hardware/architecture solutions such as HTM, NVM, RDMA, NUMA and SIMD, have been shown to be able to boost the performance of in-memory database systems significantly. Energy efficiency is also becoming attractive in the in-memory systems as DRAM contributes a relatively significant portion of the overall power consumption [289], [290], and distributed computation further exacerbates the problem. Every operational overhead that is considered negligible in disk-based systems, may become the new bottleneck in memory-based systems. Thus the removal of these legacy bottlenecks such as system calls, network stack, and cross-cacheline data layout, would contribute to a significant performance boost for in-memory systems. Furthermore, as exemplified in [117], even the implementation matters a lot in the overhead-sensitive in-memory environment.

7 Conclusions

As memory becomes the new disk, in-memory data management and processing becomes increasingly interesting for both academia and industry. Shifting the data storage layer from disks to main memory can lead to more than $100\times$ theoretical improvement in terms of response time and throughput. When data access becomes faster, every source of overhead that does not matter in traditional disk-based systems, may degrade the overall performance significantly. The shifting prompts a rethinking of the design of traditional systems, especially for databases, in the aspect of data layouts, indexes, parallelism, concurrency control, query processing, fault-tolerance, etc. Modern CPU utilization and memory-hierarchy-conscious optimization play a significant role in the design of in-memory systems, and new hardware technologies such as HTM and RDMA provide a promising opportunity to resolve problems encountered by software solutions.

In this survey, we have focused on the design principles for in-memory data management and processing, and practical techniques for designing and implementing efficient and high-performance in-memory systems. We reviewed the memory hierarchy and some advanced technologies such as NUMA and transactional memory, which provide the basis for in-memory data management and processing. In addition, we also discussed some pioneering in-memory NewSQL and NoSQL databases including cache systems, batch and online/continuous processing systems. We highlighted some promising design techniques in detail, from which we can learn the practical and concrete system design principles. This survey provides a comprehensive review of important technology in memory management and analysis of related works to date, which hopefully will be a useful resource for further memory-oriented system research.

ACKNOWLEDGMENTS

This work was supported by A*STAR project 1321202073. The authors would like to thank the anonymous reviewers, and also Bingsheng He, Eric Lo, and Bogdan Marius Tudor for their insightful comments and suggestions.

REFERENCES

[1] S. Robbins, "RAM is the new disk," *InfoQ News*, Jun. 2008.

[2] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for RAMClouds: Scalable high-performance storage entirely in dram," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, pp. 92–105, 2010.

[3] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, "Distributed data management using MapReduce," *ACM Comput. Surv.*, vol. 46, pp. 31:1–31:42, 2014.

[4] HP. (2011). Vertica systems [Online]. Available: http://www.vertica.com

[5] Hadapt Inc.. (2011). Hadapt: Sql on hadoop [Online]. Available: http://hadapt.com/

[6] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," in *Proc. VLDB Endowment*, vol. 2, pp. 1626–1629, 2009.

[7] Apache. (2008). Apache hbase [Online]. Available: http://hbase.apache.org/

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 251–264.

[9] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann, "Asterixdb: A scalable, open source BDMS," in *Proc. Very Large Database*, pp. 1905–1916, 2014.

[10] MySQL AB. (1995). Mysql: The world's most popular open source database [Online]. Available: http://www.mysql.com/

[11] Apache. (2008). Apache cassandra [Online]. Available: http://cassandra.apache.org/

[12] Oracle. (2013). Oracle database 12c [Online]. Available: https://www.oracle.com/database/index.html

[13] Neo Technology, "Neo4j - the world's leading graph database," 2007. [Online]. Available: http://www.neo4j.org/

[14] Aurelius. (2012). Titan—distributed graph database [Online]. Available: http://thinkaurelius.github.io/titan/

[15] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 31–46.

[16] Objectivity Inc. (2010). Infinitegraph [Online]. Available: http://www.objectivity.com/infinitegraph

[17] Apache. (2010). Apache Hama [Online]. Available: https://hama.apache.org

[18] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "IBM infosphere streams for scalable, real-time, intelligent transportation services," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2010, pp. 1093–1104.

[19] S. Hoffman, *Apache Flume: Distributed Log Collection for Hadoop*. Birmingham, U.K. Packt Publishing, 2013.

[20] Apache. (2005). Apache hadoop [Online]. Available: http://hadoop.apache.org/

[21] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 1151–1162.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 59–72.

[23] D. Jiang, G. Chen, B. C. Ooi, K.-L. Tan, and S. Wu, "epiC: An extensible and scalable system for processing big data," in *Proc. VLDB Endowment*, vol. 7, pp. 541–552, 2014.

[24] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "LogBase: A scalable log-structured database system in the cloud," in *Proc. VLDB Endowment*, vol. 5, pp. 1004–1015, 2012.

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, pp. 205–220, 2007.

[26] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proc. 19th ACM Symp. Operating Syst. Principles*, 2003, pp. 29–43.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.

[28] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu, "ES2: A cloud data storage system for supporting both OLTP and OLAP," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 291–302.

[29] FoundationDB. (2013). Foundationdb[Online]. Available: https://foundationdb.com

[30] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 1–14.

[31] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 1–13.

[32] B. Debnath, S. Sengupta, and J. Li, "Skimpystash: Ram space skimpy key-value store on flash-based storage," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2011, pp. 25–36.

[33] Clustrix Inc. (2006). Clustrix [Online]. Available: http://www.clustrix.com/

[34] V. Srinivasan and B. Bulkowski, "Citrusleaf: A real-time NoSQL DB which preserves acid," in *Proc. Int. Conf. Very Large Data Bases*, 2011, vol. 4, pp. 1340–1350.

[35] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots," in *IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 195–206.

[36] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: A high-performance, distributed main memory transaction processing system," *Proc. VLDB Endowment*, vol. 1, pp. 1496–1499, 2008.

[37] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL server's memory-optimized OLTP engine," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2013, pp. 1243–1254.

[38] T. Lahiri, M.-A. Neimat, and S. Folkman, "Oracle timesten: An in-memory database for enterprise applications," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 6–13, Jun. 2013.

[39] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. ACM Symp. Operating Syst. Principles*, 2013, pp. 18–32.