What's Really New With NewSQL

@ANDY_PAVLO

ISTC Big Data
Parallel Data Laboratory, Carnegie Mellon University
Carnegie Mellon Database Group

Fast

Repetitive

Small

The Last Decade of DATABASE SYSTEMS

Early 2000s – Sharding Middleware

- Custom middleware to combine multiple nodes together into a logical database.
- Route queries to correct node.

Middleware Problems

- Developers spend time writing middleware rather than working on core applications.
- Some features are implemented in application code.

Late 2000s — NoSQL

- Forgo transactional guarantees in order to achieve high-availability and high-scalability.
- Non-relational data models.

NoSQL Problems

- Developers write code to handle eventually consistent data, lack of transactions, and joins.
- Not all applications can give up strong transactional semantics.

The Rise of NewSQL Systems

Aslett White Paper

[Systems that] deliver the scalability and flexibility promised by NoSQL while retaining the support for SQL queries and/or ACID, or to improve performance for appropriate workloads.

Matt Aslett – 451 Group (April 4th, 2011)
https://www.451research.com/report-short?entityId=66963

the 451 group

451 TECHDEALMAKER	4 April 2011 – Sector IQ

How will the database incumbents respond to NoSQL and NewSQL?

Analyst: Matt Aslett

The acquisition of **MySQL AB** by **Sun Microsystems** in January 2008 appeared to signal that open source databases were on the brink of opening up a new battleground against the proprietary database giants. In announcing the deal, Sun signaled its intention to provide the support and development resources required for MySQL to challenge the established vendors in supporting mission-critical, high-performance applications on Web-based architectures. Needless to say, reality was somewhat different as Sun faced wider problems of its own and eventually succumbed to takeout by **Oracle** (Nasdaq: ORCL) in April 2009, in doing so handing ownership of the leading commercial open source database to the database heavyweight.

We had previously argued that MySQL was very much the crown jewel of the open source database world thanks to its focus on Web applications, its lightweight architecture and its fast read capabilities, which made it potentially complementary technology for all of the established database players. Additionally, if Oracle's major rivals were seeking an obvious alternative to MySQL in 2009, they were out of luck.

Just two years later, however, the database market is awash with open source databases with lightweight architectures targeted at Web applications. Not only have the likes of **Monty Program** and **SkySQL** emerged to provide alternative support for MySQL and its forks, but there are also a large number of products available under the banner of NoSQL, which emerged in mid-2009 as an umbrella term for a loosely affiliated collection of non-relational database projects. We have also seen the emergence of what we have termed 'NewSQL' database offerings, with companies promising to deliver the scalability and flexibility promised by NoSQL while retaining the support for SQL queries and/or ACID (atomicity, consistency, isolation and durability), or to improve performance for appropriate workloads to the extent that the advanced scalability promised by some NoSQL databases becomes irrelevant.

From MySQL to NoSQL

Despite being a good match for many read-intensive applications, MySQL does not provide predictable performance at scale, particularly with a few writes thrown into the mix. The memcached distributed memory object-caching system can be used – and has been widely adopted – to improve performance but does not provide any persistence and lacks consistency. To some extent, the rise of NoSQL has been driven by the inadequacies of

Stonebraker Article

SQL as the primary interface.
ACID support for transactions
Non-locking concurrency control.
High per-node performance.
Shared-nothing architecture.

COMMUNICATIONS OF THE ACM
TRUSTED INSIGHTS FOR COMPUTING'S LEADING PROFESSIONALS

Home » Blogs » BLOG@CACM » New SQL: An Alternative to NoSQL and Old SQL for New... » Full Text

BLOG@CACM

New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps

Michael Stonebraker

June 16, 2011

Historically, Online Transaction Processing (OLTP) was performed by customers submitting traditional transactions (order something, withdraw money, cash a check, etc.) to a relational DBMS. Large enterprises might have dozens to hundreds of these systems. Invariably, enterprises wanted to consolidate the information in these OLTP systems for business analysis, cross selling, or some other purpose. Hence, Extract-Transform-and-Load (ETL) products were used to convert OLTP data to a common format and load it into a data warehouse. Data warehouse activity rarely shared machine resources with OLTP because of lock contention in the DBMS and because business intelligence (BI) queries were so resource-heavy that they got in the way of timely responses to transactions.

This combination of a collection of OLTP systems, connected to ETL, and connected to one or more data warehouses is the gold standard in enterprise computing. I will term it "Old OLTP." By and large, this activity was supported by the traditional RDBMS vendors. In the past I have affectionately called them "the elephants"; in this posting I refer to them as "Old SQL."

As noted by most pundits, "the Web changes everything," and I have noticed a very different collection of OLTP requirements that are emerging for Web properties, which I will term "New OLTP." These sites seem to be driven by two customer requirements:

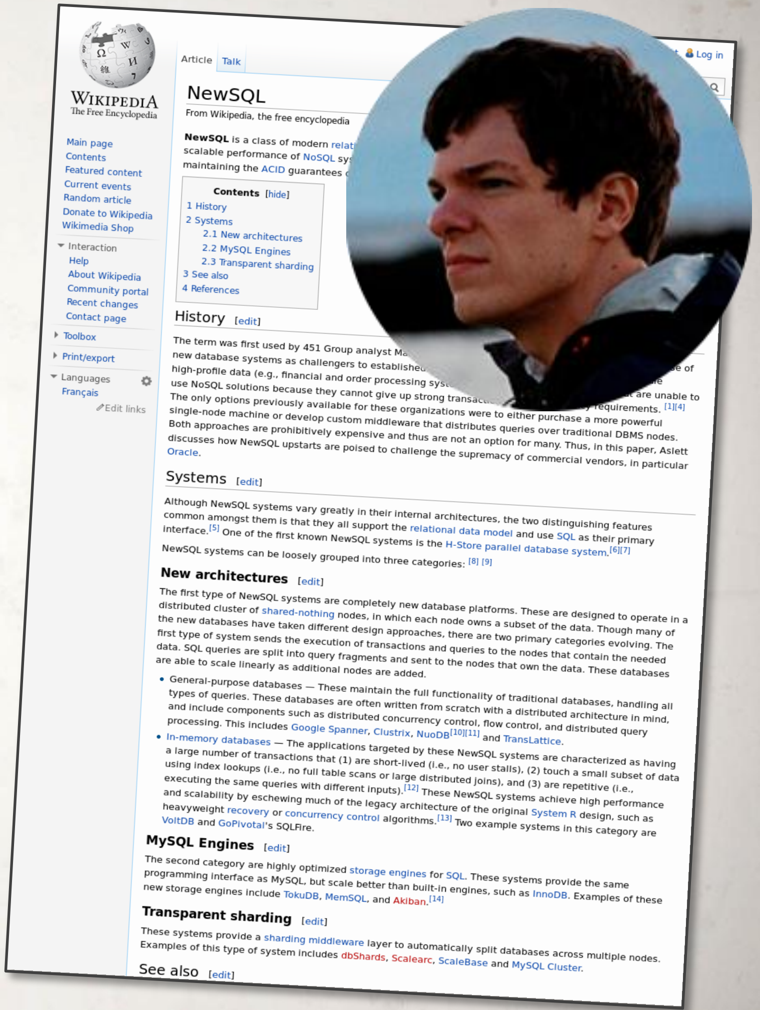
The need for far more OLTP throughput. Consider new Web-based applications such as multi-player games, social networking sites, and online gambling networks. The aggregate number of interactions per second is skyrocketing for the successful Web properties in this category. In addition, the explosive growth of smartphones has created a market for applications that use the phone as a geographic sensor and provide location-based services. Again, successful

Mike Stonebraker – Blog@CACM (June 16th, 2011)
http://cacm.acm.org/blogs/blog-cacm/109710

Wikipedia Article

A class of modern relational database systems that provide the same scalable performance of NoSQL systems for OLTP workloads while still maintaining the ACID guarantees of a traditional database system.

Wikipedia (October 19th, 2013)
http://en.wikipedia.org/wiki/NewSQL



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Toolbox
Print/export
Languages
Français
Edit links

Article Talk

Log in

NewSQL

From Wikipedia, the free encyclopedia

NewSQL is a class of modern relat... scalable performance of NoSQL sy... maintaining the ACID guarantees ...

Contents [hide]
1 History
2 Systems
2.1 New architectures
2.2 MySQL Engines
2.3 Transparent sharding
3 See also
4 References

History [edit]

The term was first used by 451 Group analyst Ma... new database systems as challengers to establishe... high-profile data (e.g., financial and order processing sys... use NoSQL solutions because they cannot give up strong transac... requirements.[1][4] The only options previously available for these organizations were to either purchase a more powerful single-node machine or develop custom middleware that distributes queries over traditional DBMS nodes. Both approaches are prohibitively expensive and thus are not an option for many. Thus, in this paper, Aslett discusses how NewSQL upstarts are poised to challenge the supremacy of commercial vendors, in particular Oracle.

Systems [edit]

Although NewSQL systems vary greatly in their internal architectures, the two distinguishing features common amongst them is that they all support the relational data model and use SQL as their primary interface.[5] One of the first known NewSQL systems is the H-Store parallel database system.[6][7]
NewSQL systems can be loosely grouped into three categories:[8][9]

New architectures [edit]

The first type of NewSQL systems are completely new database platforms. These are designed to operate in a distributed cluster of shared-nothing nodes, in which each node owns a subset of the data. Though many of the new databases have taken different design approaches, there are two primary categories evolving. The first type of system sends the execution of transactions and queries to the nodes that contain the needed data. SQL queries are split into query fragments and sent to the nodes that own the data. These databases are able to scale linearly as additional nodes are added.

- General-purpose databases — These maintain the full functionality of traditional databases, handling all types of queries. These databases are often written from scratch with a distributed architecture in mind, and include components such as distributed concurrency control, flow control, and distributed query processing. This includes Google Spanner, Clustrix, NuoDB[10][11] and TransLattice.
- In-memory databases — The applications targeted by these NewSQL systems are characterized as having a large number of transactions that (1) are short-lived (i.e., no user stalls), (2) touch a small subset of data using index lookups (i.e., no full table scans or large distributed joins), and (3) are repetitive (i.e., executing the same queries with different inputs).[12] These NewSQL systems achieve high performance and scalability by eschewing much of the legacy architecture of the original System R design, such as heavyweight recovery or concurrency control algorithms.[13] Two example systems in this category are VoltDB and GoPivotal's SQLFire.

MySQL Engines [edit]

The second category are highly optimized storage engines for SQL. These systems provide the same programming interface as MySQL, but scale better than built-in engines, such as InnoDB. Examples of these new storage engines include TokuDB, MemSQL, and Akiban.[14]

Transparent sharding [edit]

These systems provide a sharding middleware layer to automatically split databases across multiple nodes. Examples of this type of system includes dbShards, Scalearc, ScaleBase and MySQL Cluster.

See also [edit]


SCALABILITY
HIGH
(Many Nodes)
NoSQL
NEWSQL
TRADITIONAL
LOW
(One Node)
WEAK
(None/Limited)
GUARANTEES
STRONG
(ACID)

New Design

- NuoDB
- VoltDB
- SAP HANA
- deep db
- memsql
- Clustrix
- Google Spanner

MySQL Engines

- ScaleDB
- Tokutek

Middleware

- ScaleBase
- ScaleArc
- dbShards

New Design

NUODB
VOLTDB
SAP HANA
deepdb
memsql
Clustrix
Google Spanner

MySQL Engines

ScaleDB
Tokutek

Middleware

ScaleBase
ScaleArc
dbShards

Distributed Concurrency Control

Distributed Concurrency Control

SAP HANA

memsql

Clustrix

Google Spanner

NUODB

VOLTDB

Distributed Concurrency Control

SAP HANA

memsql

Clustrix

Google Spanner

NUODB

VOLTDB

Distributed Concurrency Control

SAP HANA

memsql

Clustrix

Google Spanner

NUODB

VOLTDB

Distributed Concurrency Control

SAP HANA

memsql

Clustrix

Google Spanner

NUODB

VOLTDB

Concurrency Control in Distributed Database Systems

PHILIP A. BERNSTEIN AND NATHAN GOODMAN

Computer Corporation of America, Cambridge, Massachusetts 02139

In this paper we survey, consolidate, and present the state of the art in distributed database concurrency control. The heart of our analysis is a decomposition of the concurrency control problem into two major subproblems: read-write and write-write synchronization. We describe a series of synchronization techniques for solving each subproblem and show how to combine these techniques into algorithms for solving the entire concurrency control problem. Such algorithms are called "concurrency control methods." We describe 48 principal methods, including all practical algorithms that have appeared in the literature plus several new ones. We concentrate on the structure and correctness of concurrency control algorithms. Issues of performance are given only secondary treatment.

Keywords and Phrases: concurrency control, deadlock, distributed database management systems, locking, serializability, synchronization, timestamp ordering, timestamps, two-phase commit, two-phase locking

CR Categories: 4.33, 4.35

INTRODUCTION

The Concurrency Control Problem

Concurrency control is the activity of coordinating concurrent accesses to a database in a multiuser database management system (DBMS). Concurrency control permits users to access a database in a multiprogrammed fashion while preserving the illusion that each user is executing alone on a dedicated system. The main technical difficulty in attaining this goal is to prevent database updates performed by one user from interfering with database retrievals and updates performed by another. The concurrency control problem is exacerbated in a distributed DBMS (DDBMS) because (1) users may access data stored in many different computers in a distributed system, and (2) a concurrency control mechanism at one computer cannot instantaneously know about interactions at other computers.

Concurrency control has been actively investigated for the past several years, and the problem for nondistributed DBMSs is well understood. A broad mathematical theory has been developed to analyze the problem, and one approach, called *two-phase locking*, has been accepted as a standard solution. Current research on nondistributed concurrency control is focused on evolutionary improvements to two-phase locking, detailed performance analysis and optimization, and extensions to the mathematical theory.

Distributed concurrency control, by contrast, is in a state of extreme turbulence. More than 20 concurrency control algorithms have been proposed for DDBMSs, and several have been, or are being, implemented. These algorithms are usually complex, hard to understand, and difficult to prove correct (indeed, many are incorrect). Because they are described in different terminologies and make different assumptions

Phil Bernstein, et al. – Computing Surveys (June 1981)
http://dl.acm.org/citation.cfm?id=356846

Main Memory Systems

SAP HANA

VOLTDB

memsql

IMPLEMENTATION TECHNIQUES FOR MAIN MEMORY DATABASE SYSTEMS

David J DeWitt[1], Randy H Katz[2], Frank Olken[3], Leonard D Shapiro[4], Michael R Stonebraker[2], David Wood[2]

[1] Computer Sciences Department, University of Wisconsin
[2] EECS Department, University of California at Berkeley
[3] CSAM Department, Lawrence Berkeley Laboratory
[4] Department of Computer Science, North Dakota State University

ABSTRACT With the availability of very large, relatively inexpensive main memories, it is becoming possible keep large databases resident in main memory In this paper we consider the changes necessary to permit a relational database system to take advantage of large amounts of main memory We evaluate AVL vs B+-tree access methods for main memory databases, hash-based query processing strategies vs sort-merge, and study recovery issues when most or all of the database fits in main memory As expected, B+-trees are the preferred storage mechanism unless more than 80-90% of the database fits in main memory A somewhat surprising result is that hash based query processing strategies are advantageous for large memory situations

Key Words and Phrases Main Memory Databases, Access Methods, Join Algorithms, Access Planning, Recovery Mechanisms

1. Introduction

Throughout the past decade main memory prices have plummetted and are expected to continue to do so At the present time, memory for super-minicomputers such as the VAX 11/780 costs approximately $1,500 a megabyte By 1990, 1 megabit memory chips will be commonplace and should further reduce prices by another order of magnitude Thus, in 1990 a gigabyte of memory should cost less than $200,000 If 4 megabit memory chips are available, the price might be as low as $50,000

With the availability of larger amounts of main memory, it becomes possible to contemplate the storage of databases as main memory objects In fact, IMS Fast Path [DATE82] has supported such databases for some time In this paper we consider the changes that might be needed to a relational database system if most (or all) of a relation(s) is (are) resident in main memory

In Section 2, the performance of alternative access methods for main memory database systems are considered Algorithms for relational database operators in this environment are presented and evaluated in Section 3 In Section 4, we describe how access planning will be affected by the availability of large amounts of main memory for query processing Section 5 discusses recovery in memory resident databases Our conclusions and suggestions for future research are contained in Section 6

This research was partially supported by the National Science Foundation under grants MCS82-01860, MCS82-01870, by the Department of Energy under contracts #DE-AC02-81ER10920, #DE-AC03-76SF00098, #W-7405-ENG-48, and by the Air Force Office of Scientific Research under Grant 83-0021

2. Access Methods for Memory Resident Databases

The standard access method for data on disk is the B+-tree [COME79], providing both random and sequential key access A B+-tree is specially designed to provide fast access to disk-resident data and makes fundamental use of the page size of the device On the other hand, if a keyed relation is known to reside in main memory, then an AVL (or other binary) tree organization may be a better choice In this section we analyze the performance of both structure for a relation R with the following characteristics

||R||	number of tuples in relation R
K	width of the key for R in bytes
L	width of a tuple in bytes
P	page size in bytes
4	size of a pointer in bytes

We have analyzed two cases of interest The first is the cost of retrieving a single tuple using a random key value An example of this type of query is

retrieve (emp salary) where emp name = "Jones"

The second case analyzed is the cost of reading N records sequentially Consider the query

retrieve (emp salary, emp name) where emp name = "J*"

which requests data on all employees whose names begin with J To execute this query, the database system would locate the first employee with a name beginning with J and then read sequentially This second case analyzes the sequential access portion of such a command

For both cases (random and sequential access), there are two costs that are specific to the access method

|page reads|	the number of pages read to execute the query
|comparisons|	the number of record comparisons required to isolate the particular data of interest

The number of comparisons is indicative of the CPU time required to process the command while the number of page reads approximates the I/O costs

To compare the performance AVL and B+-trees, we propose the following cost function

cost = Z * |page-reads| + |comparisons|

Since a page read consumes perhaps 2000 instructions of operating system overhead and 30 milliseconds of elapsed time while a comparison can easily be done in 200, we expect realistic values of Z to be in the range of 10 to 30 Later in the section we will use several values in this range

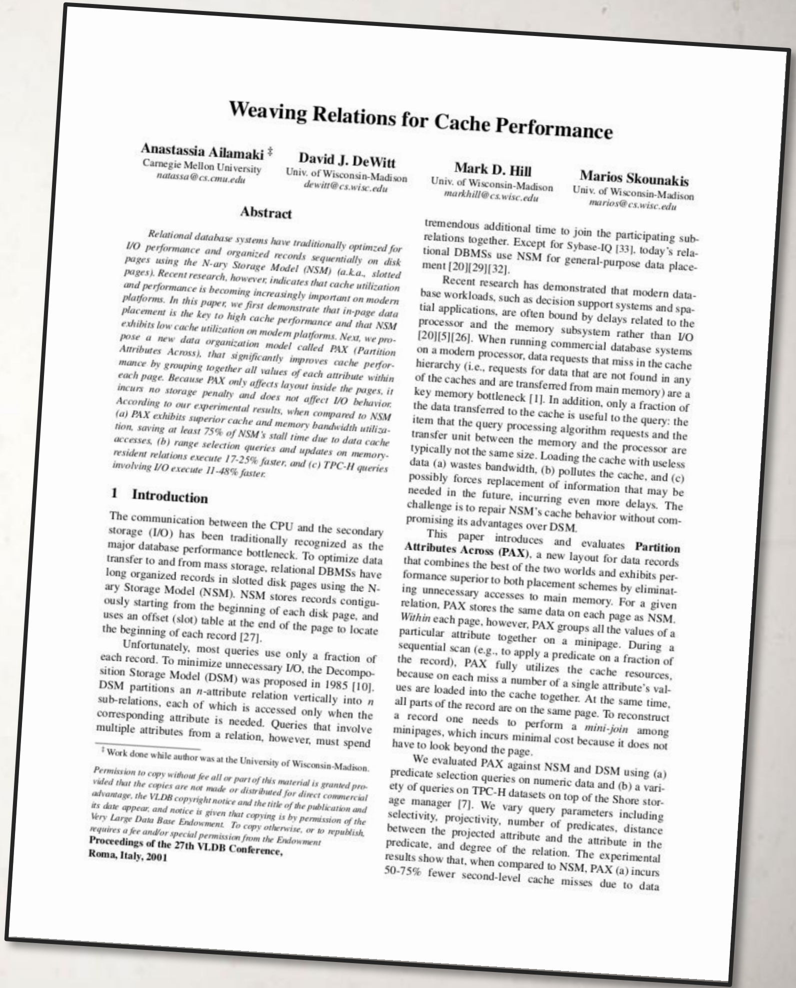
Moreover, it is possible (although not very likely) that an AVL-tree comparison will be cheaper than a B+-tree comparison The reasoning is that the B+-tree record must be located within a page while an AVL tree does not contain any page structure and records can be directly located Consequently, we assume that an AVL-tree comparison costs Y times a B+-tree comparison for some

1

David DeWitt, et al. – SIGMOD (1984)
http://dl.acm.org/citation.cfm?id=602261

Hybrid Architectures

Hybrid Architectures



Weaving Relations for Cache Performance

Anastassia Ailamaki ‡
Carnegie Mellon University
natassa@cs.cmu.edu

David J. DeWitt
Univ. of Wisconsin-Madison
dewitt@cs.wisc.edu

Mark D. Hill
Univ. of Wisconsin-Madison
markhill@cs.wisc.edu

Marios Skounakis
Univ. of Wisconsin-Madison
marios@cs.wisc.edu

Abstract

Relational database systems have traditionally optimzed for I/O performance and organized records sequentially on disk pages using the N-ary Storage Model (NSM) (a.k.a., slotted pages). Recent research, however, indicates that cache utilization and performance is becoming increasingly important on modern platforms. In this paper, we first demonstrate that in-page data placement is the key to high cache performance and that NSM exhibits low cache utilization on modern platforms. Next, we propose a new data organization model called PAX (Partition Attributes Across), that significantly improves cache performance by grouping together all values of each attribute within each page. Because PAX only affects layout inside the pages, it incurs no storage penalty and does not affect I/O behavior. According to our experimental results, when compared to NSM (a) PAX exhibits superior cache and memory bandwidth utilization, saving at least 75% of NSM's stall time due to data cache accesses, (b) range selection queries and updates on memory-resident relations execute 17-25% faster, and (c) TPC-H queries involving I/O execute 11-48% faster.

1 Introduction

The communication between the CPU and the secondary storage (I/O) has been traditionally recognized as the major database performance bottleneck. To optimize data transfer to and from mass storage, relational DBMSs have long organized records in slotted disk pages using the N-ary Storage Model (NSM). NSM stores records contiguously starting from the beginning of each disk page, and uses an offset (slot) table at the end of the page to locate the beginning of each record [27].

Unfortunately, most queries use only a fraction of each record. To minimize unnecessary I/O, the Decomposition Storage Model (DSM) was proposed in 1985 [10]. DSM partitions an n-attribute relation vertically into n sub-relations, each of which is accessed only when the corresponding attribute is needed. Queries that involve multiple attributes from a relation, however, must spend tremendous additional time to join the participating sub-relations together. Except for Sybase-IQ [33], today's relational DBMSs use NSM for general-purpose data placement [20][29][32].

Recent research has demonstrated that modern database workloads, such as decision support systems and spatial applications, are often bound by delays related to the processor and the memory subsystem rather than I/O [20][5][26]. When running commercial database systems on a modern processor, data requests that miss in the cache hierarchy (i.e., requests for data that are not found in any of the caches and are transferred from main memory) are a key memory bottleneck [1]. In addition, only a fraction of the data transferred to the cache is useful to the query: the item that the query processing algorithm requests and the transfer unit between the memory and the processor are typically not the same size. Loading the cache with useless data (a) wastes bandwidth, (b) pollutes the cache, and (c) possibly forces replacement of information that may be needed in the future, incurring even more delays. The challenge is to repair NSM's cache behavior without compromising its advantages over DSM.

This paper introduces and evaluates **Partition Attributes Across (PAX)**, a new layout for data records that combines the best of the two worlds and exhibits performance superior to both placement schemes by eliminating unnecessary accesses to main memory. For a given relation, PAX stores the same data on each page as NSM. *Within* each page, however, PAX groups all the values of a particular attribute together on a minipage. During a sequential scan (e.g., to apply a predicate on a fraction of the record), PAX fully utilizes the cache resources, because on each miss a number of a single attribute's values are loaded into the cache together. At the same time, all parts of the record are on the same page. To reconstruct a record one needs to perform a *mini-join* among minipages, which incurs minimal cost because it does not have to look beyond the page.

We evaluated PAX against NSM and DSM using (a) predicate selection queries on numeric data and (b) a variety of queries on TPC-H datasets on top of the Shore storage manager [7]. We vary query parameters including selectivity, projectivity, number of predicates, distance between the projected attribute and the attribute in the predicate, and degree of the relation. The experimental results show that, when compared to NSM, PAX (a) incurs 50-75% fewer second-level cache misses due to data

‡ Work done while author was at the University of Wisconsin-Madison.

Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001

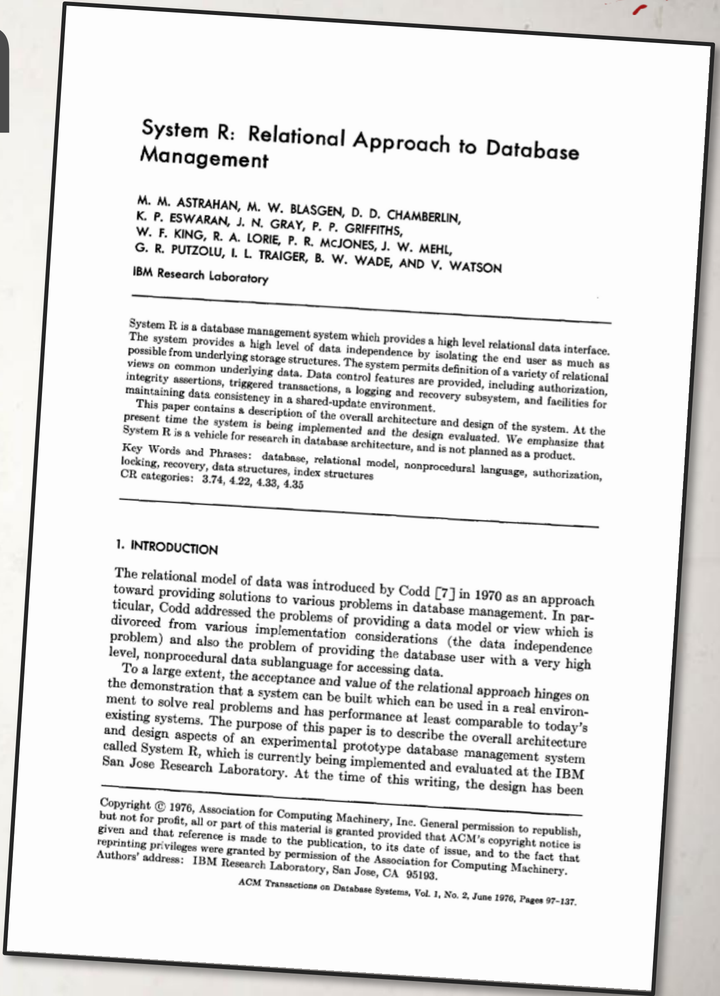
Anastassia Ailamaki, et al. – VLDB (2001)
http://dl.acm.org/citation.cfm?id=672367

Query Code Compilation

memsql

Google Spanner

Query Code Compilation



System R: Relational Approach to Database Management

M. M. ASTRAHAN, M. W. BLASGEN, D. D. CHAMBERLIN,
K. P. ESWARAN, J. N. GRAY, P. P. GRIFFITHS,
W. F. KING, R. A. LORIE, P. R. MCJONES, J. W. MEHL,
G. R. PUTZOLU, I. L. TRAIGER, B. W. WADE, AND V. WATSON
IBM Research Laboratory

System R is a database management system which provides a high level relational data interface. The system provides a high level of data independence by isolating the end user as much as possible from underlying storage structures. The system permits definition of a variety of relational views on common underlying data. Data control features are provided, including authorization, integrity assertions, triggered transactions, a logging and recovery subsystem, and facilities for maintaining data consistency in a shared-update environment.

This paper contains a description of the overall architecture and design of the system. At the present time the system is being implemented and the design evaluated. We emphasize that System R is a vehicle for research in database architecture, and is not planned as a product.

Key Words and Phrases: database, relational model, nonprocedural language, authorization, locking, recovery, data structures, index structures
CR categories: 3.74, 4.22, 4.33, 4.35

1. INTRODUCTION

The relational model of data was introduced by Codd [7] in 1970 as an approach toward providing solutions to various problems in database management. In particular, Codd addressed the problems of providing a data model or view which is divorced from various implementation considerations (the data independence problem) and also the problem of providing the database user with a very high level, nonprocedural data sublanguage for accessing data.

To a large extent, the acceptance and value of the relational approach hinges on the demonstration that a system can be built which can be used in a real environment to solve real problems and has performance at least comparable to today's existing systems. The purpose of this paper is to describe the overall architecture and design aspects of an experimental prototype database management system called System R, which is currently being implemented and evaluated at the IBM San Jose Research Laboratory. At the time of this writing, the design has been

Authors' address: IBM Research Laboratory, San Jose, CA 95193.

M. M. Astrahan, et al. – ACM TODS (1976)
http://dl.acm.org/citation.cfm?id=320457

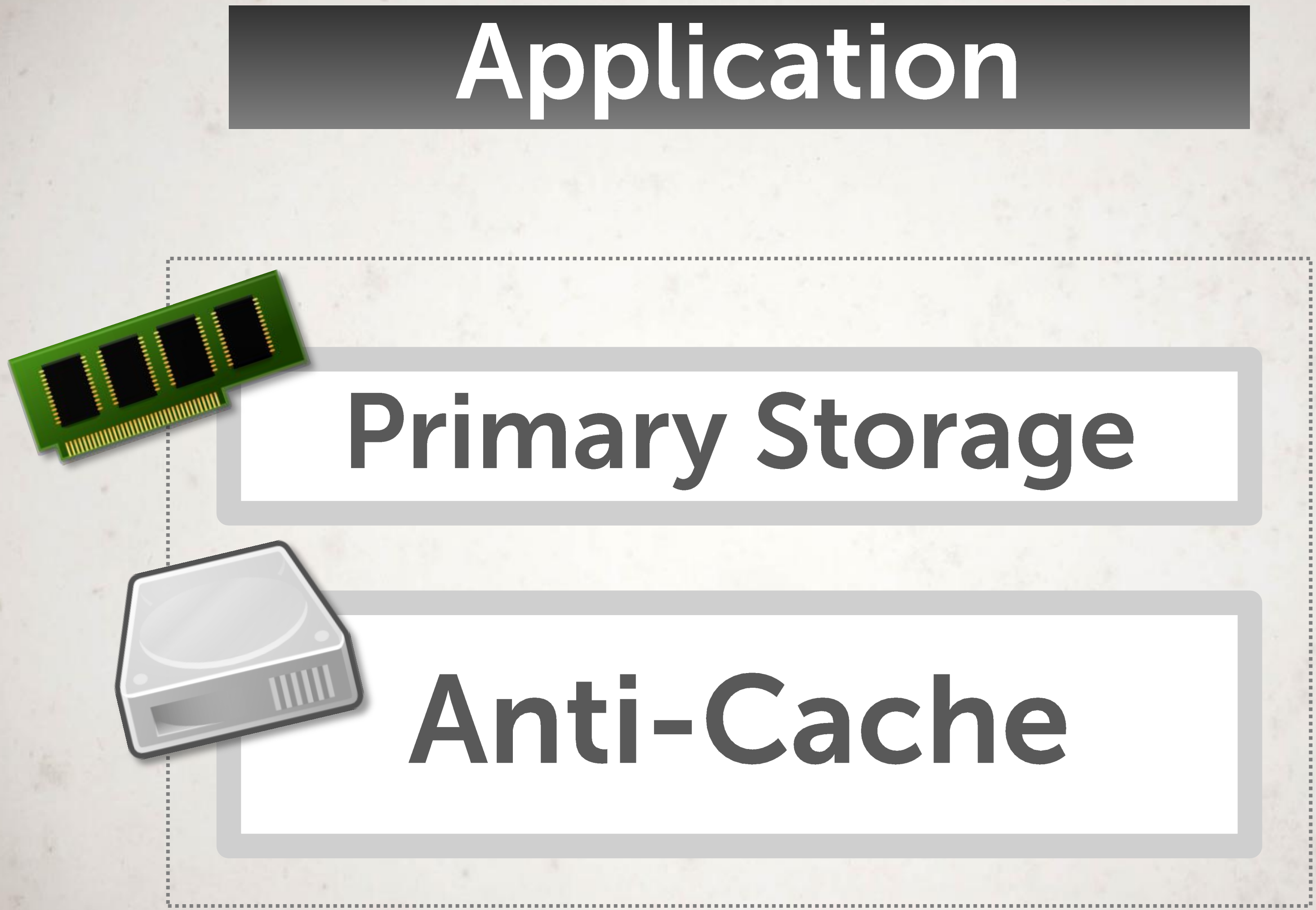
Recap

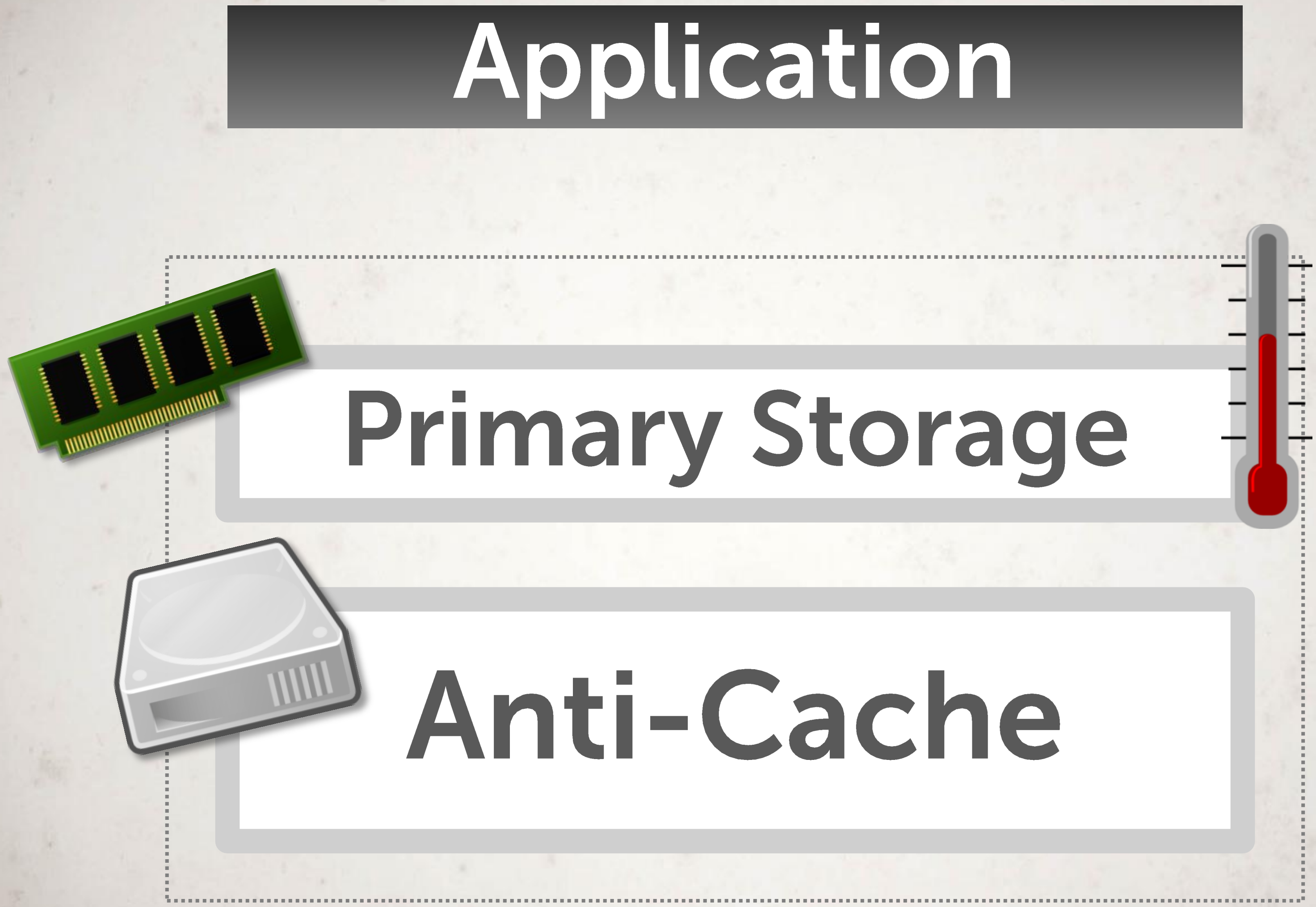
- Distributed Concurrency Control
- Main Memory Stores
- Hybrid Architectures
- Query Code Compilation

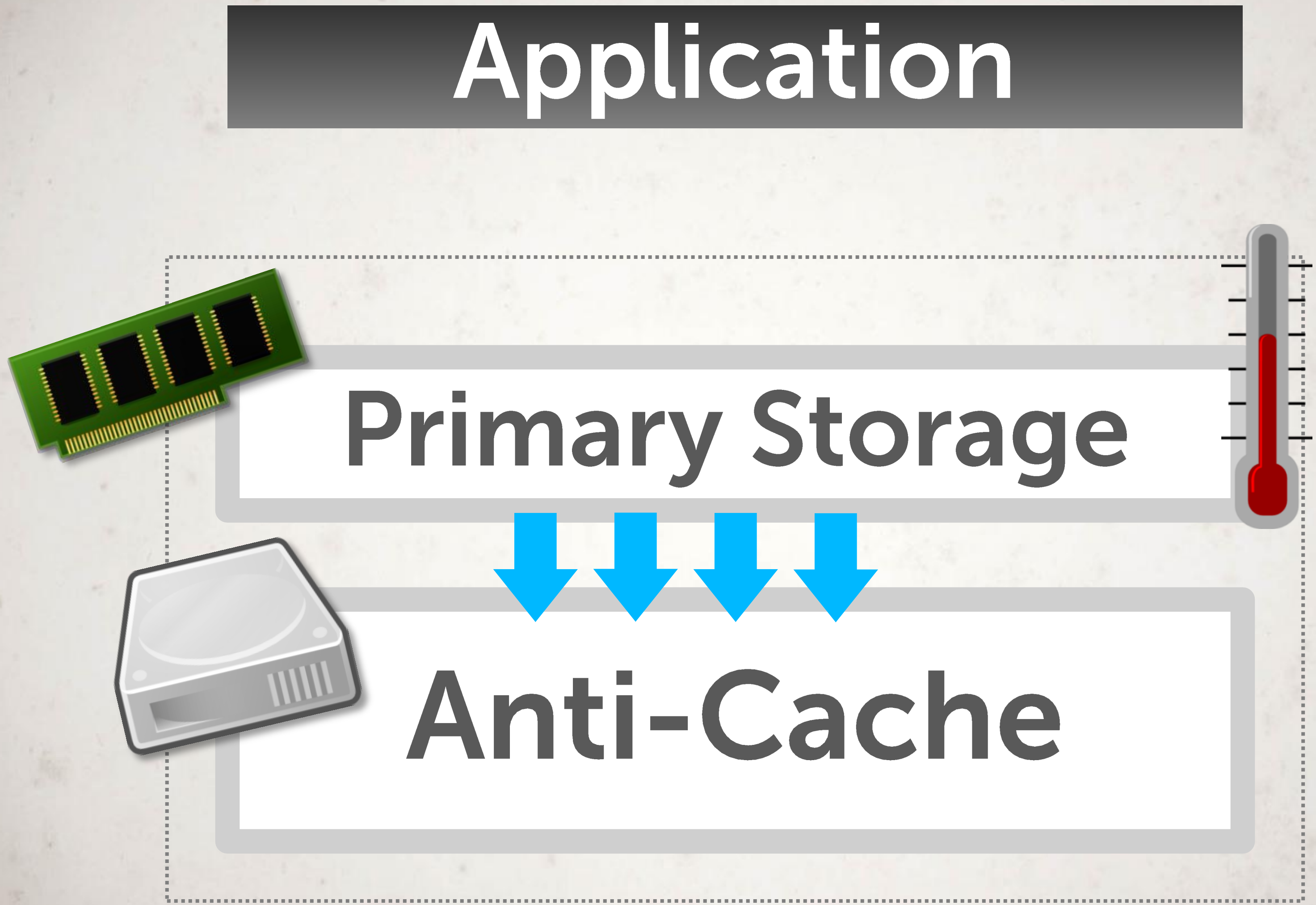
The Future of
OLTP DBMS RESEARCH

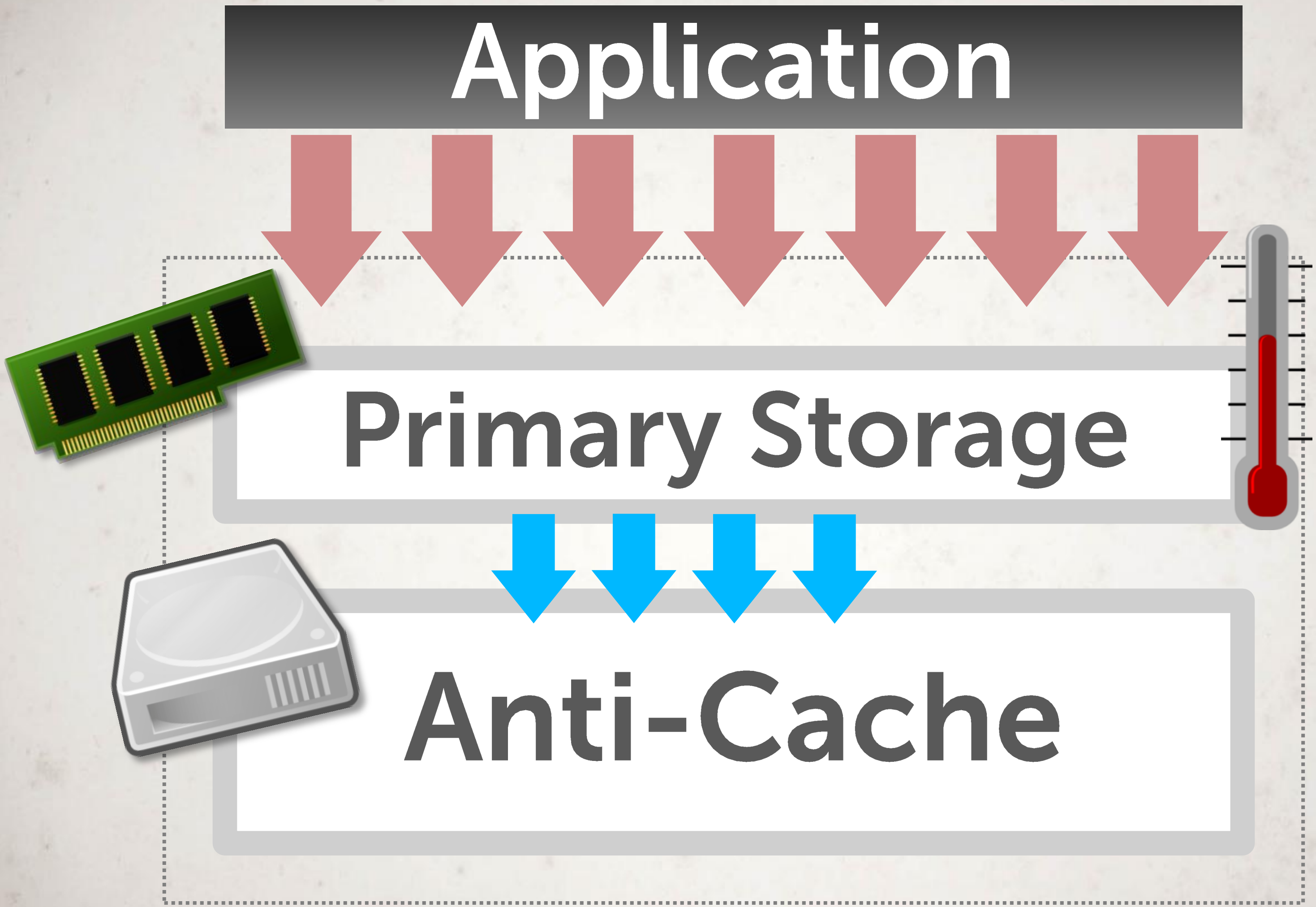
Nearly Solved Problems

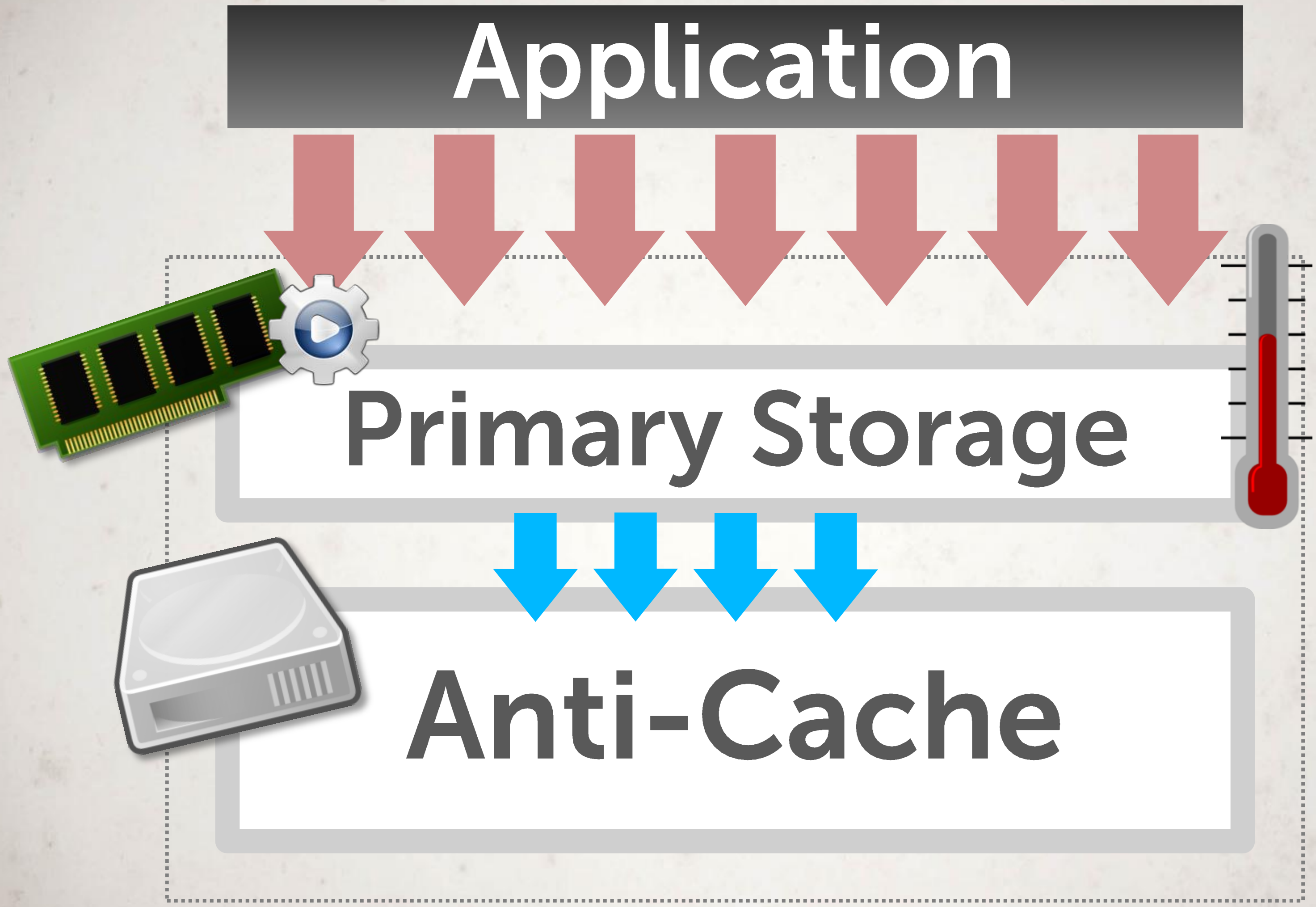
- Fine-grain Elasticity
- Hybrid Architectures
- Larger-than Memory DBs

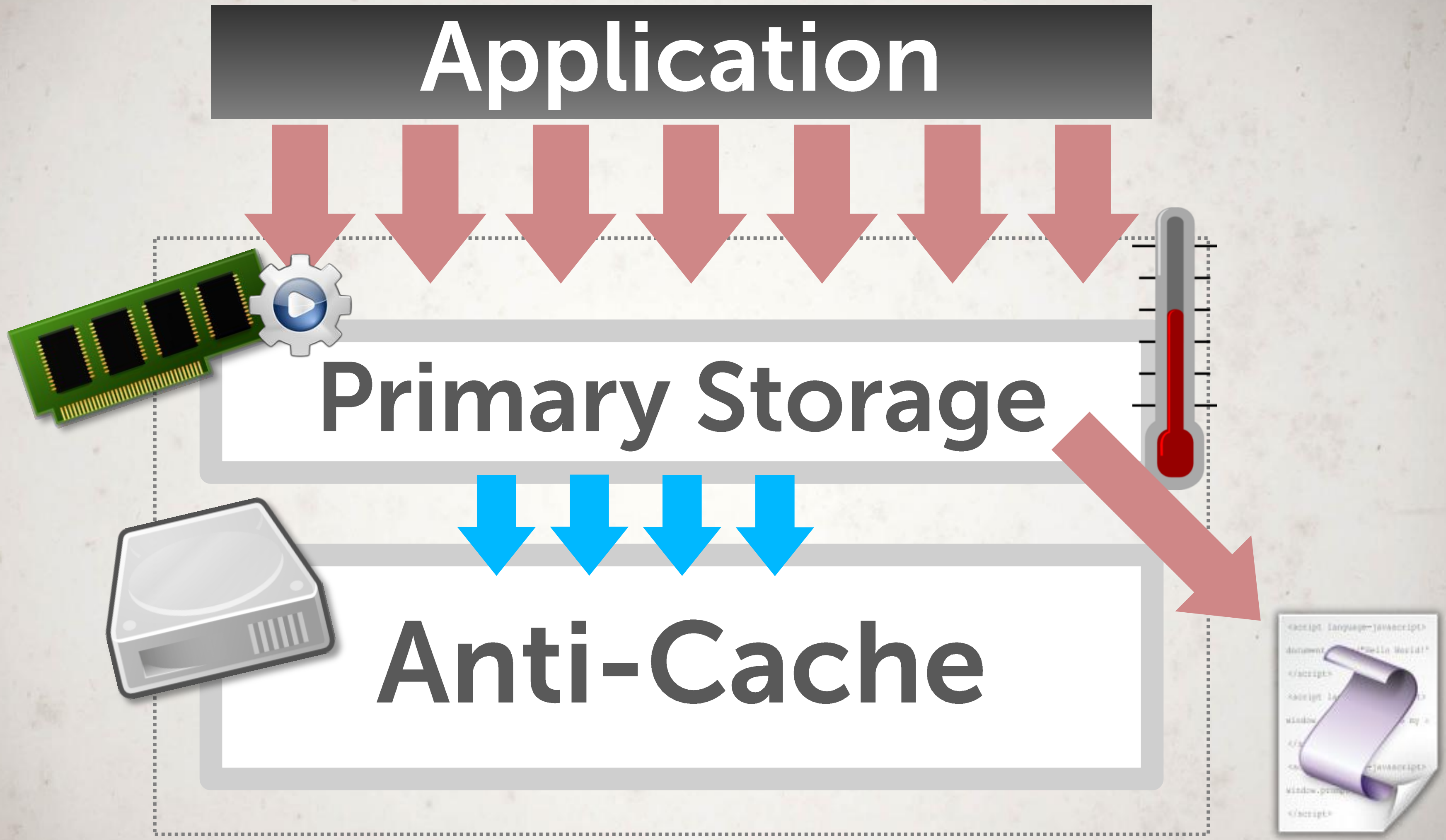

Application
Primary Storage
Anti-Cache

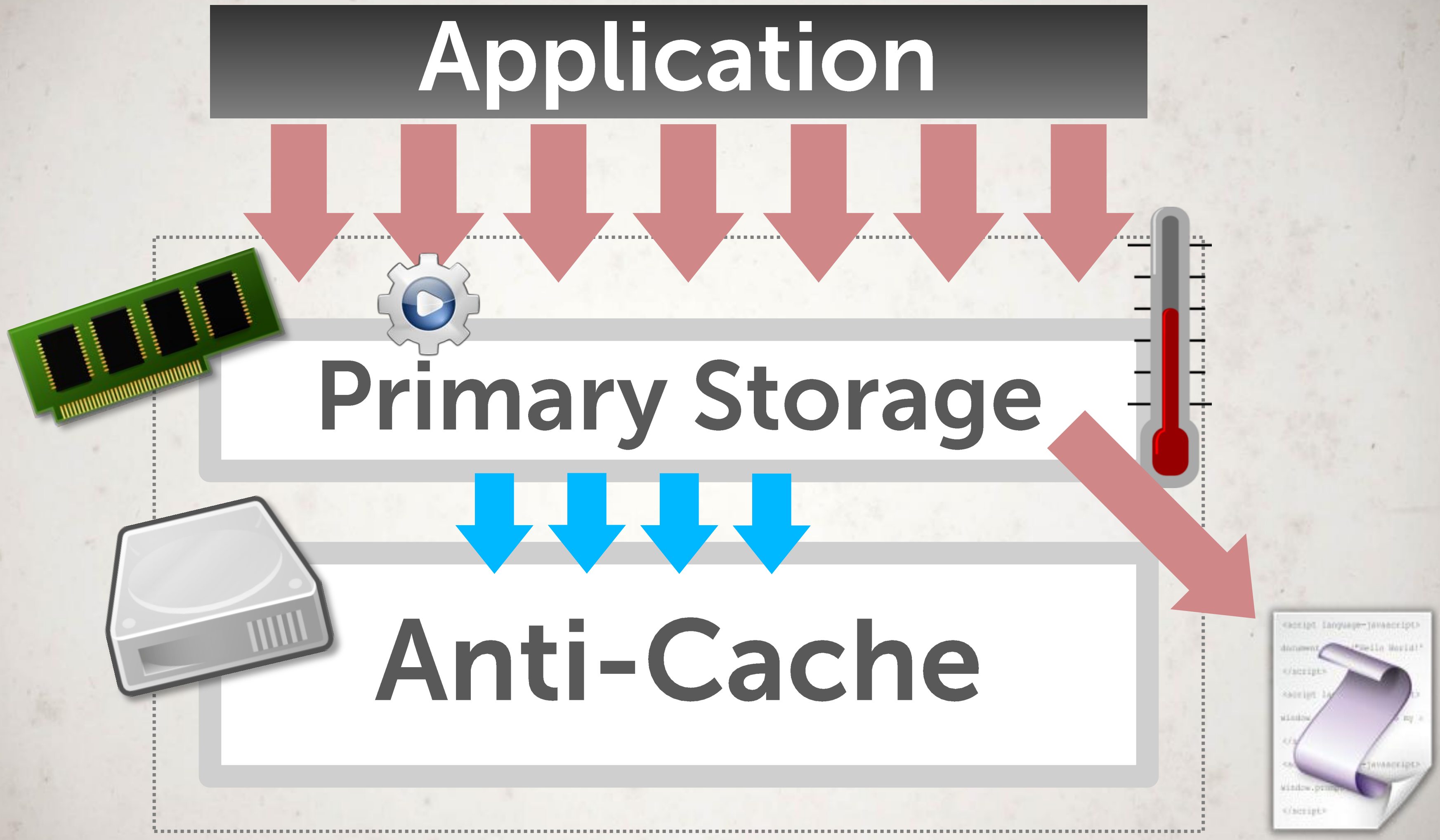

Application
Primary Storage
Anti-Cache

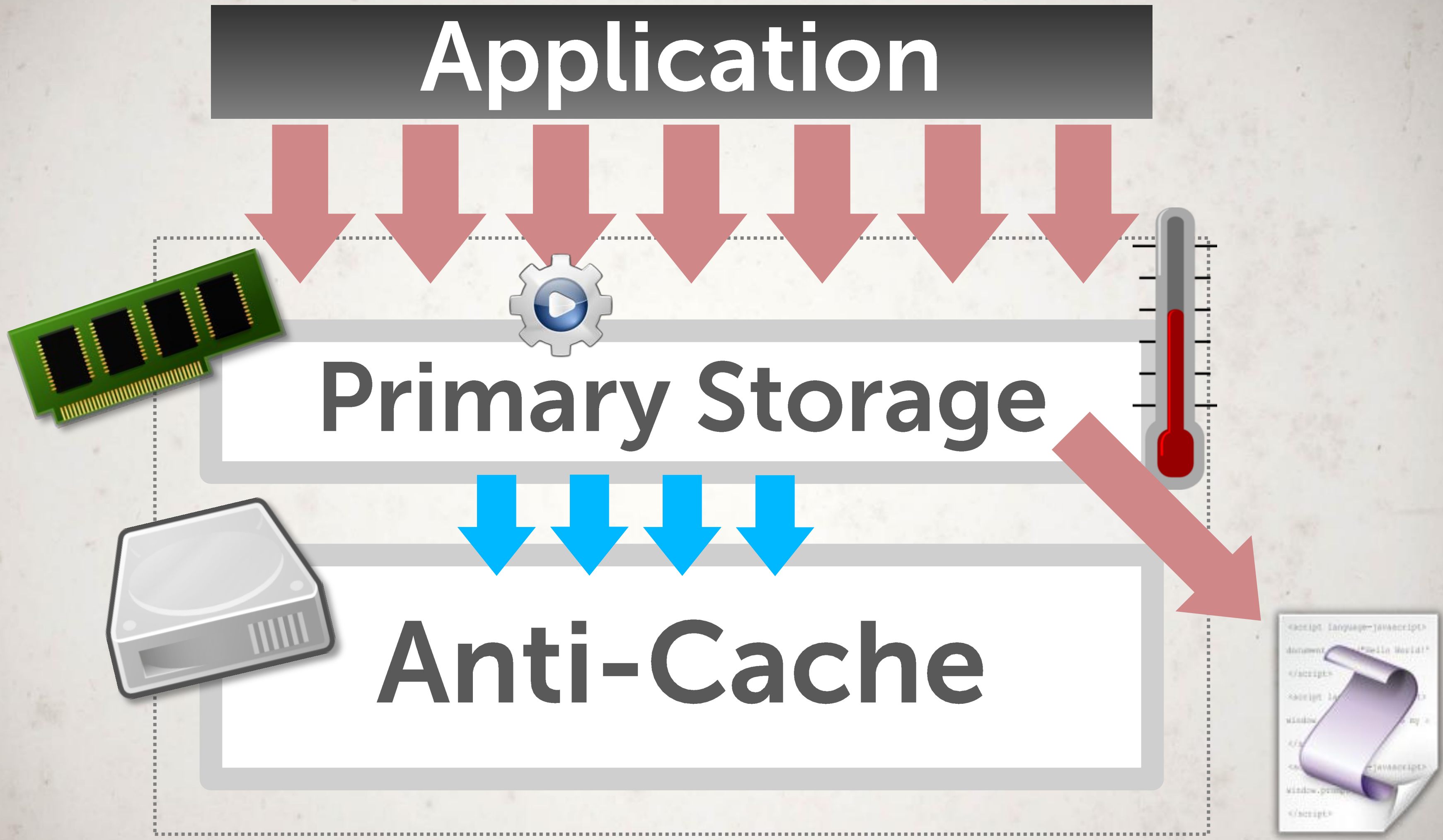

Application
Primary Storage
Anti-Cache


Application
Primary Storage
Anti-Cache

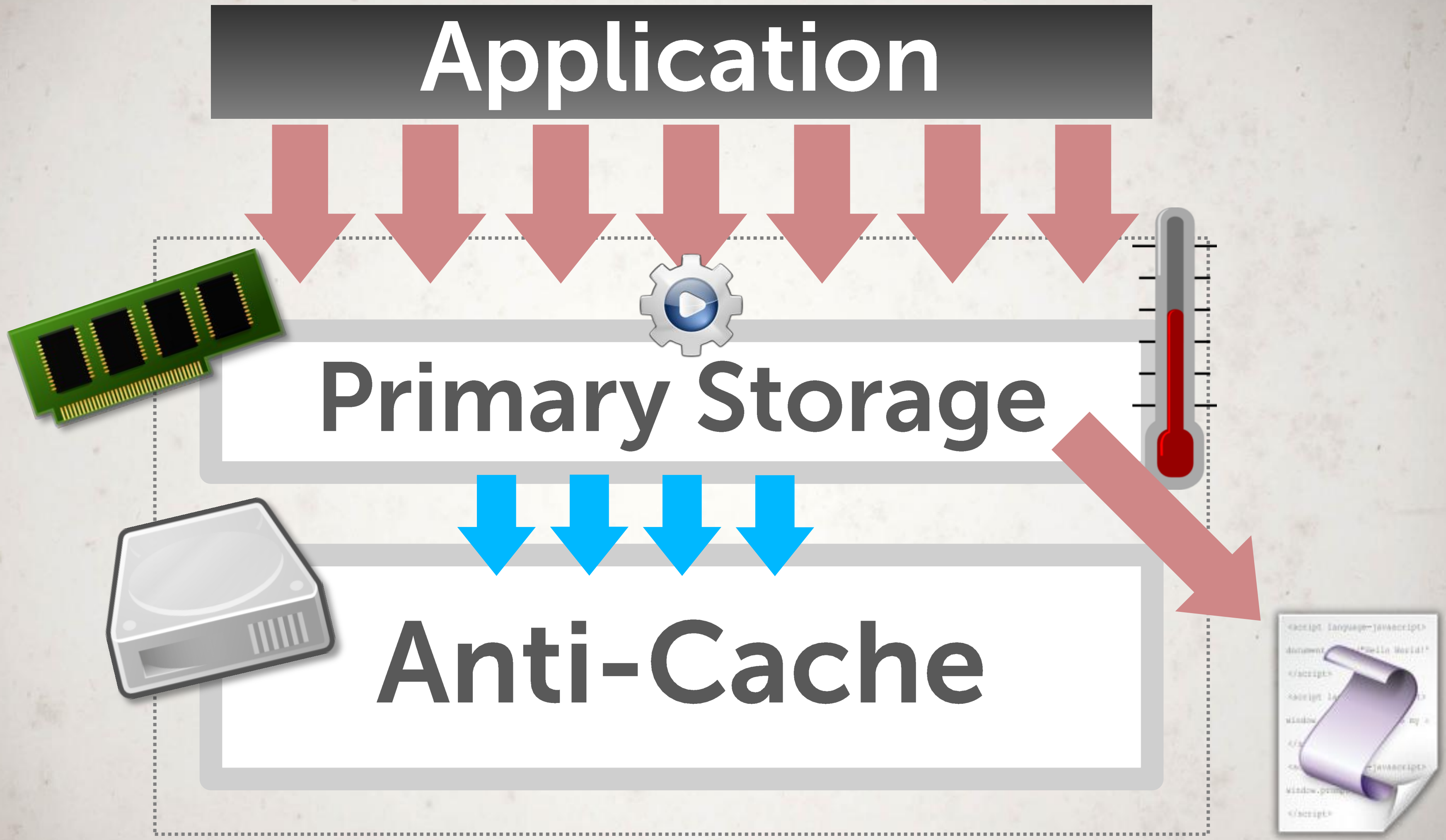

Application
Primary Storage
Anti-Cache

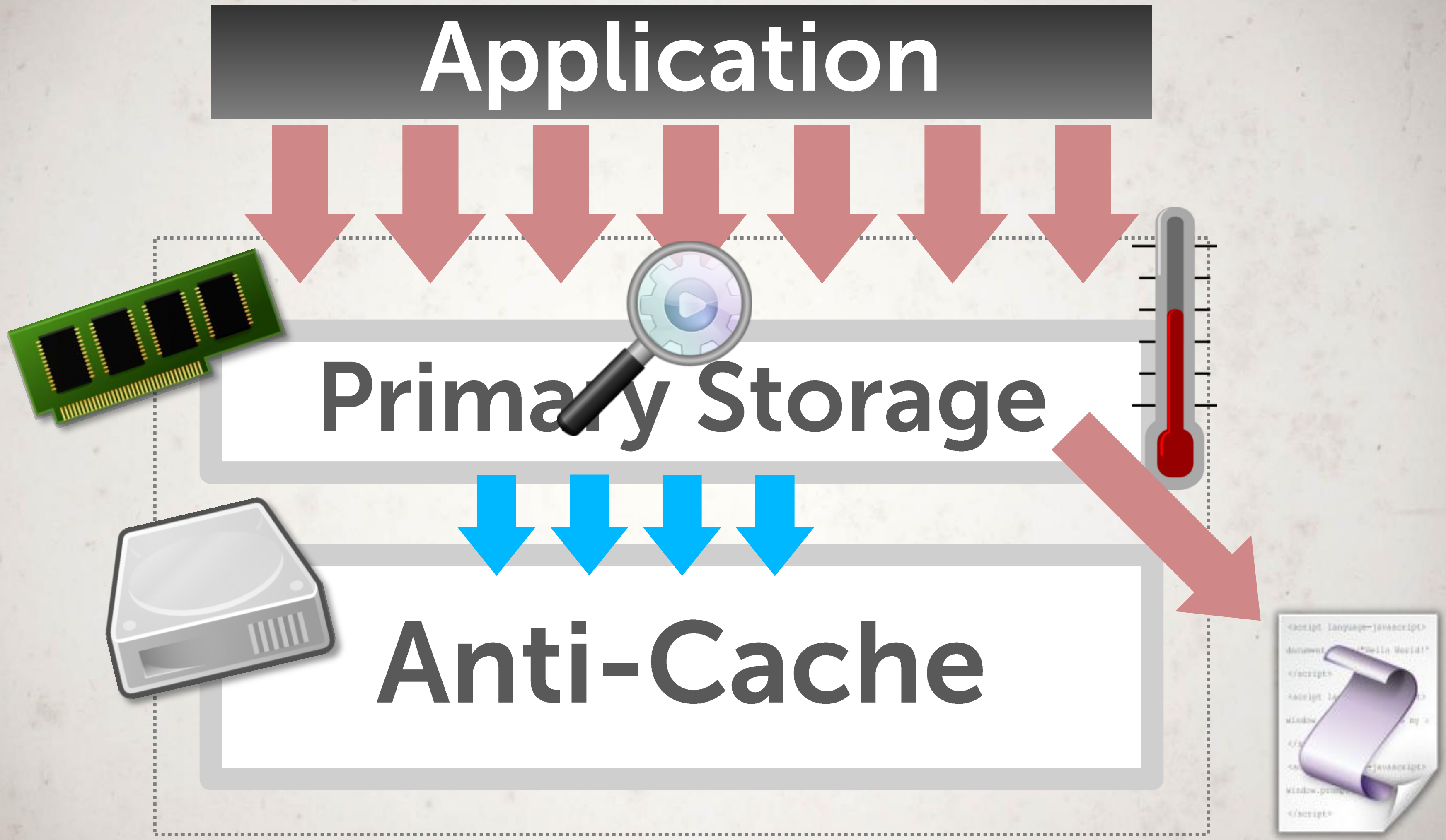

Application
Primary Storage
Anti-Cache

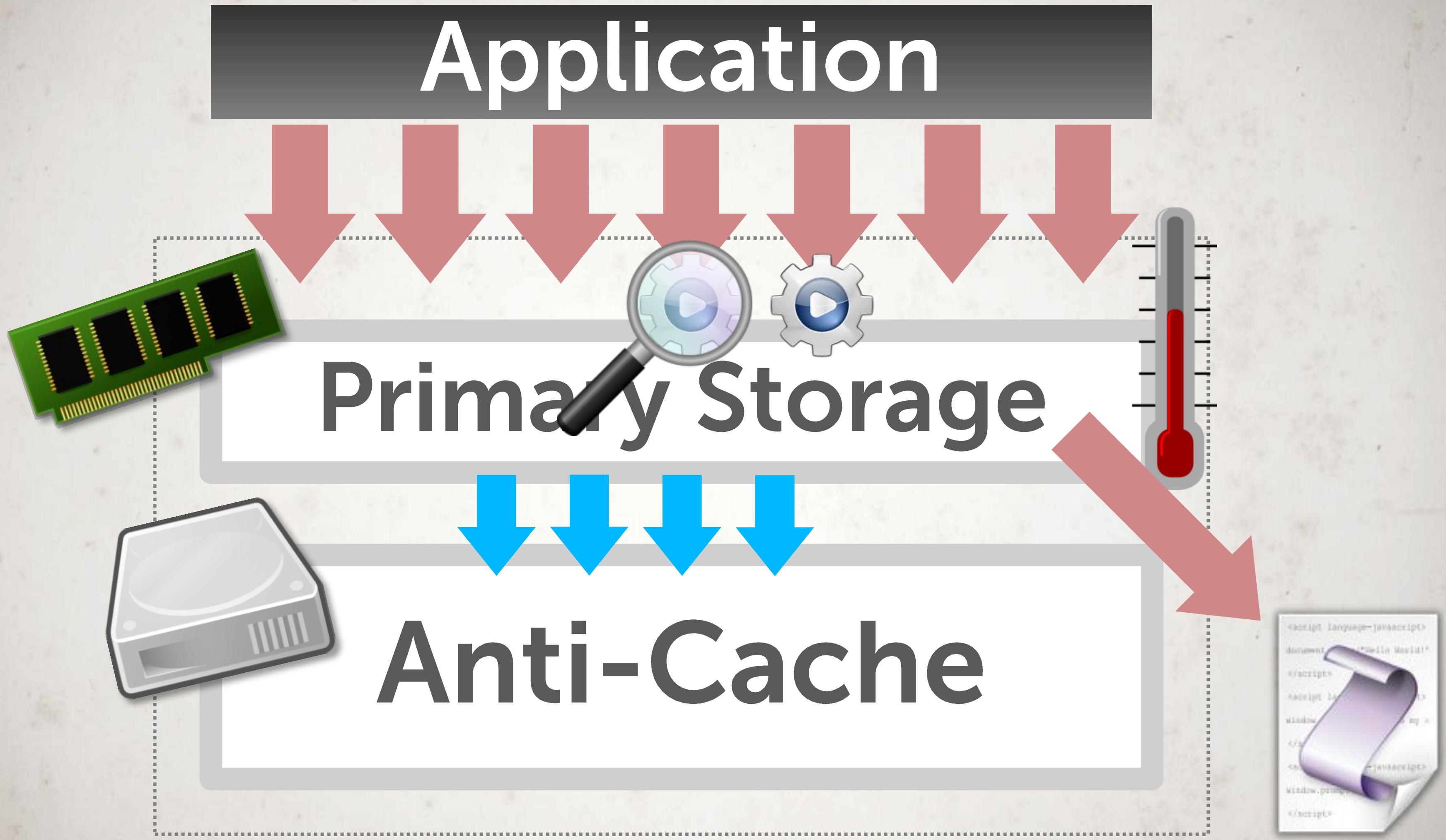

Application
Primary Storage
Anti-Cache

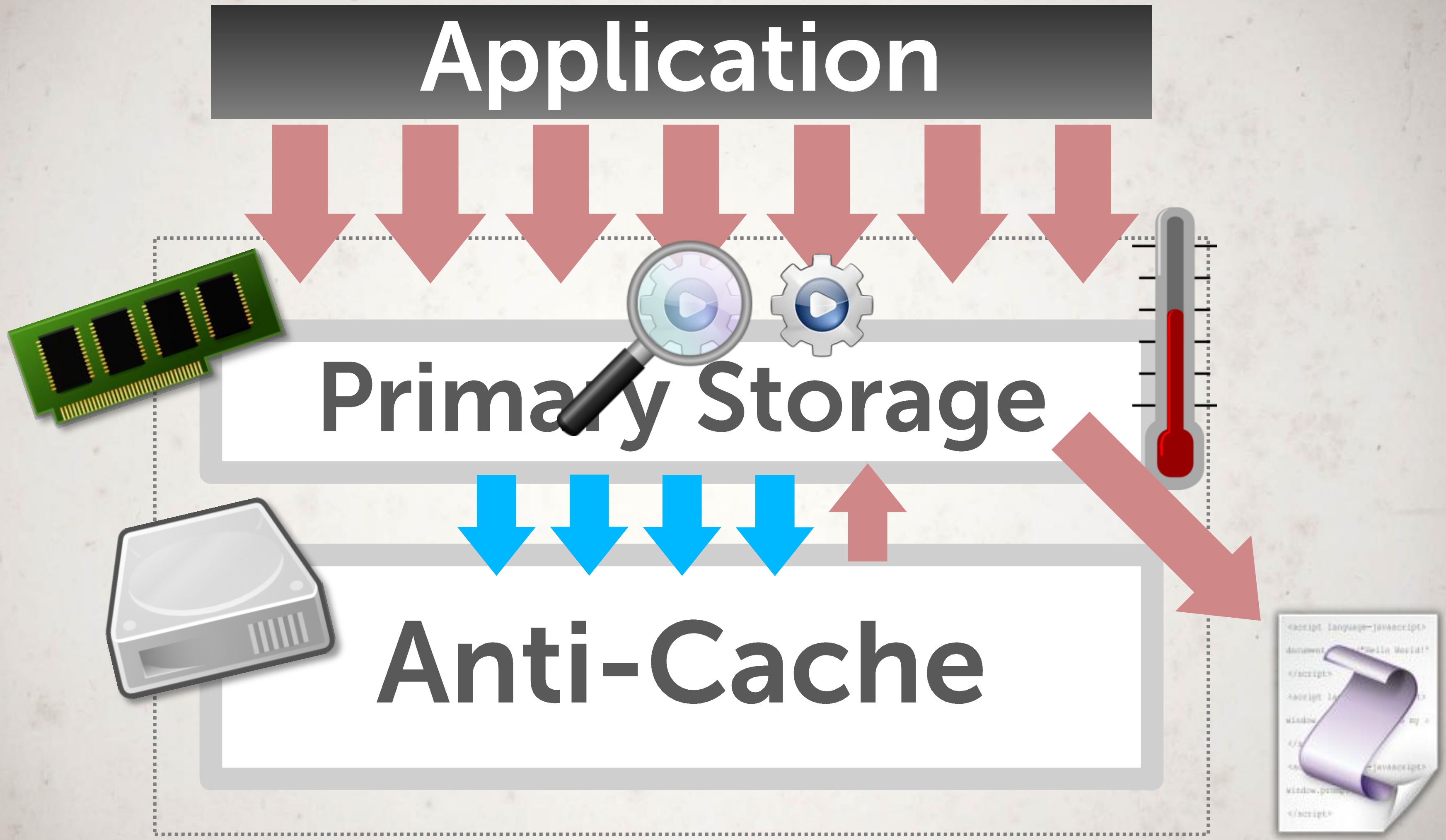


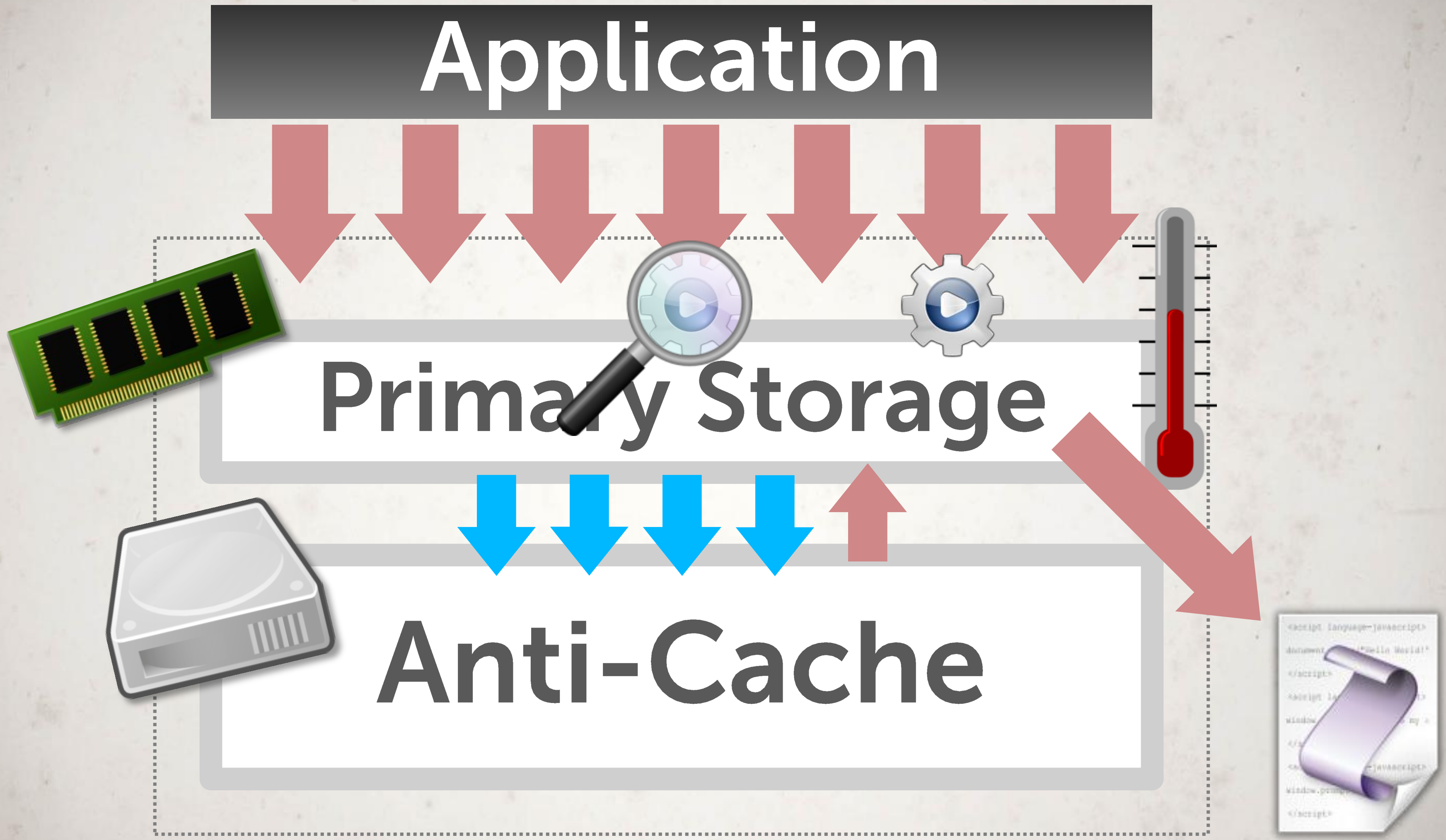
Application
Primary Storage
Anti-Cache

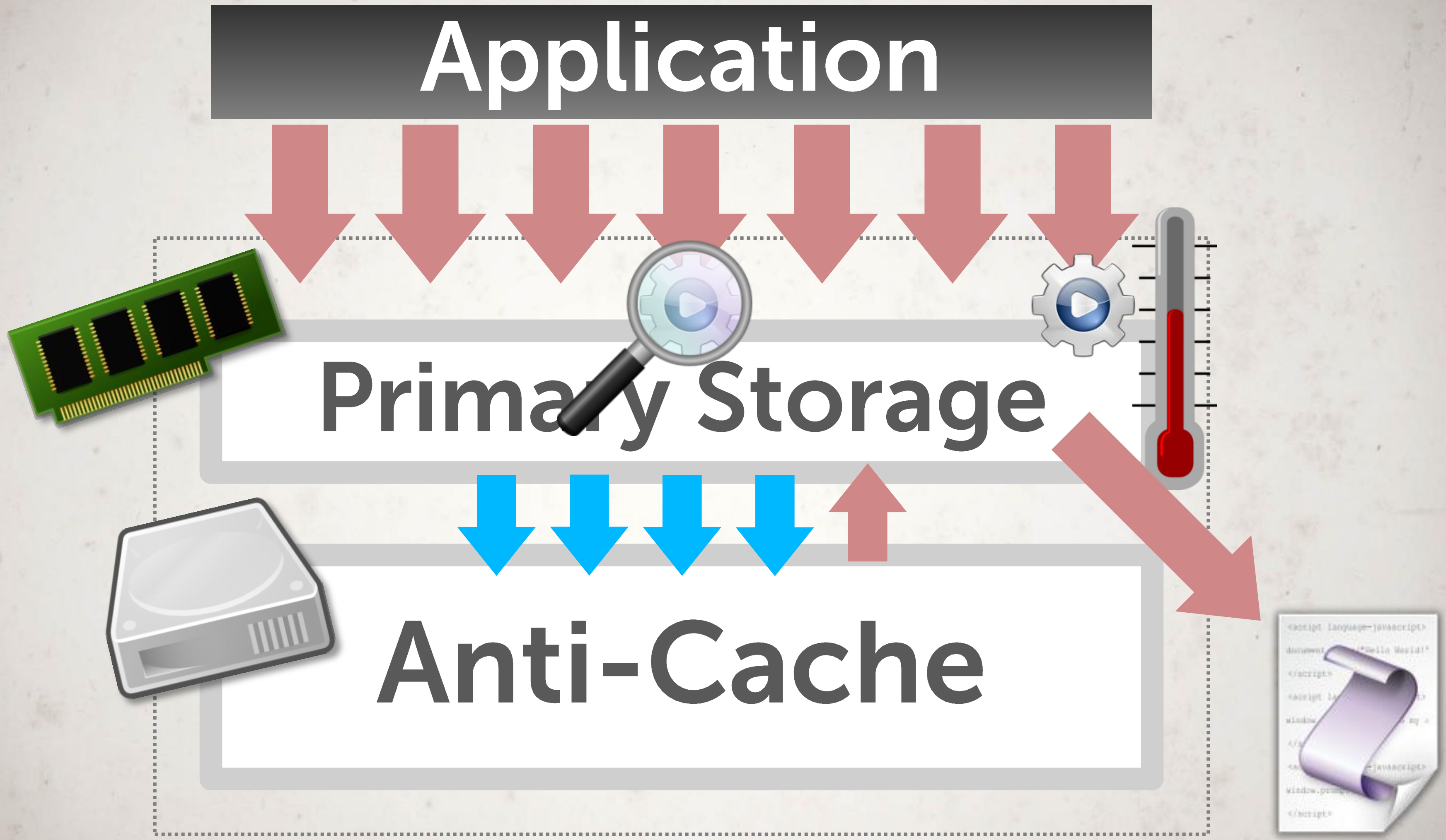

Application
Primary Storage
Anti-Cache

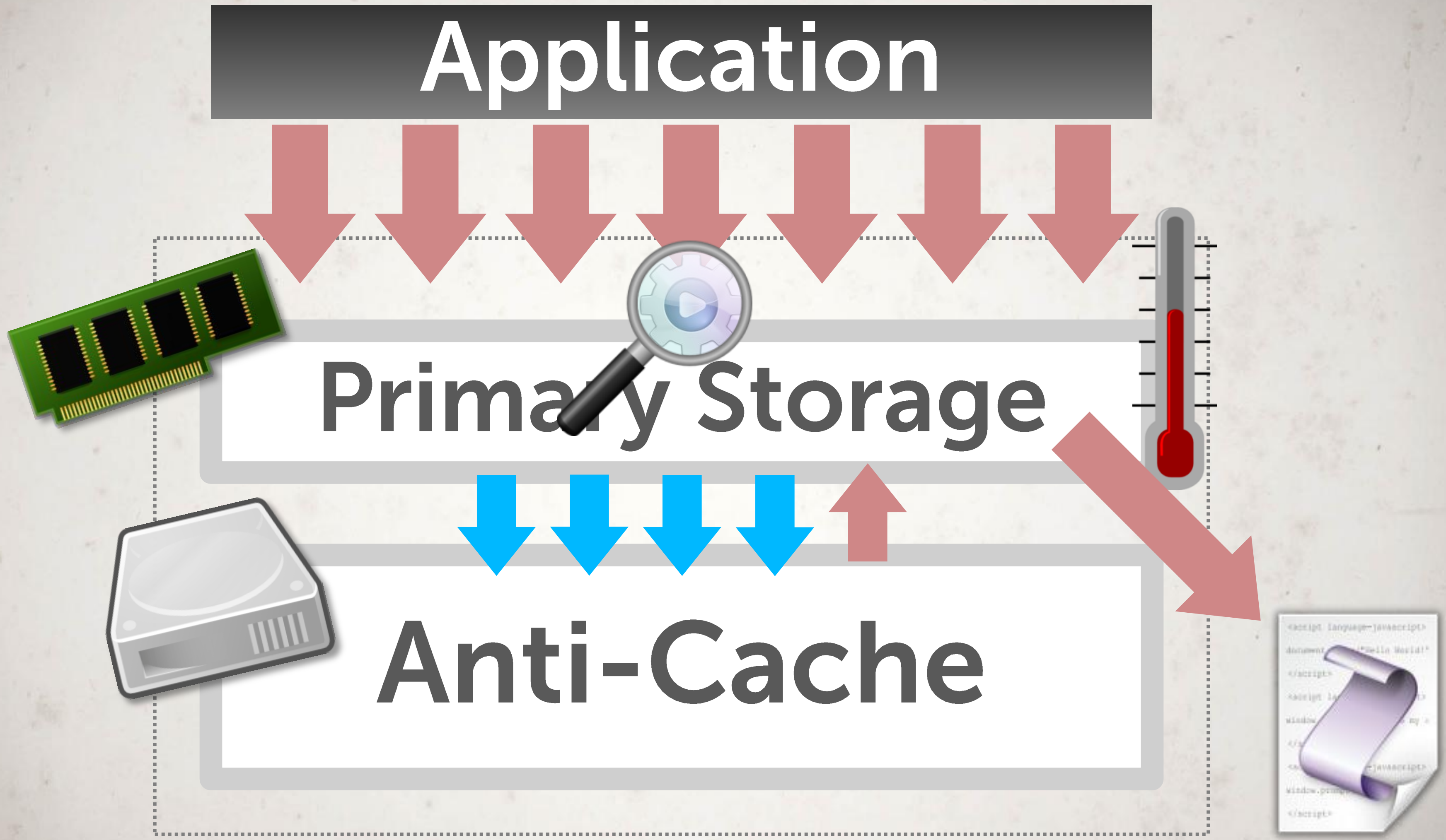

Application
Primary Storage
Anti-Cache

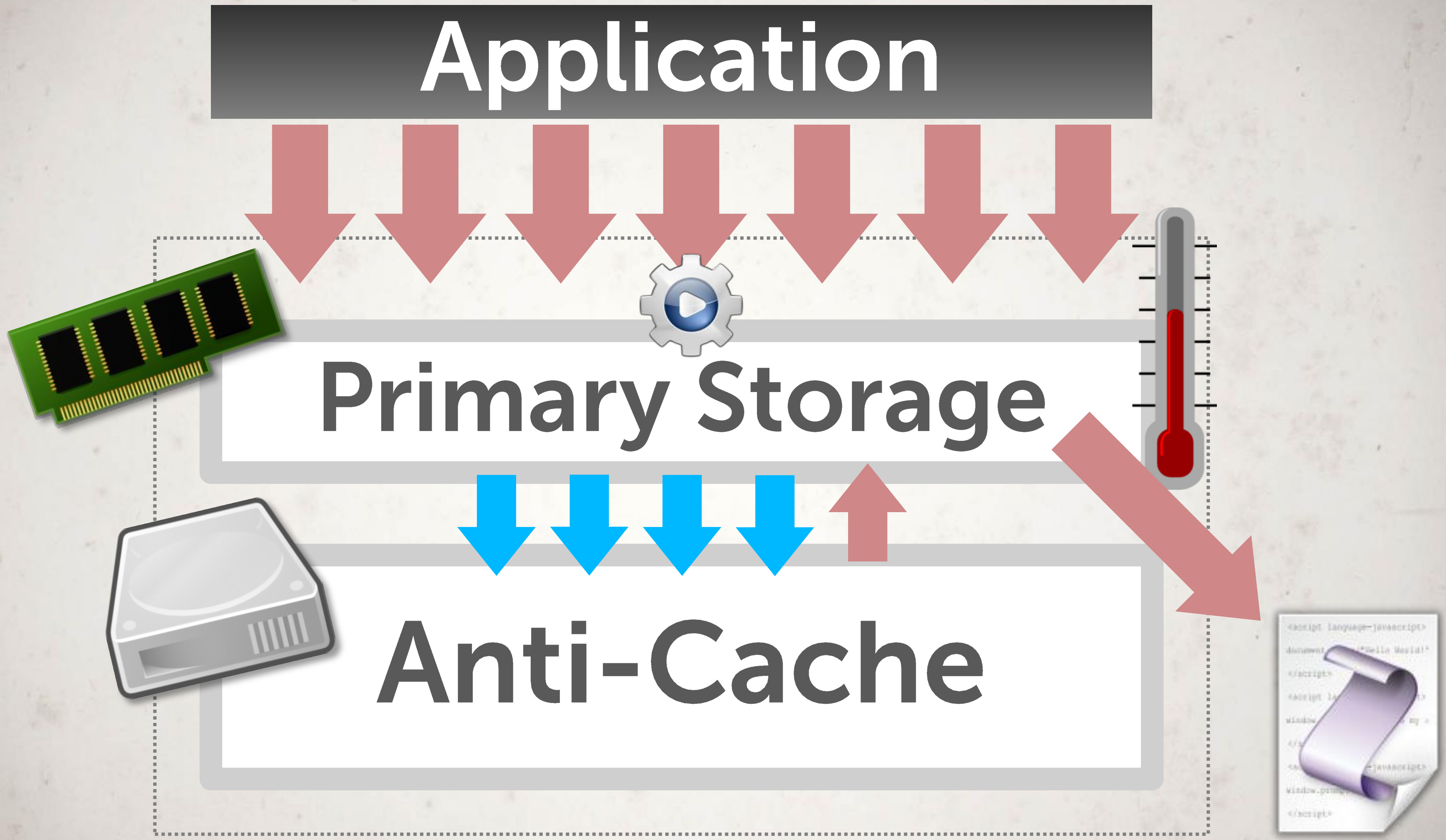

Application
Primary Storage
Anti-Cache


Application
Primary Storage
Anti-Cache


Application
Primary Storage
Anti-Cache

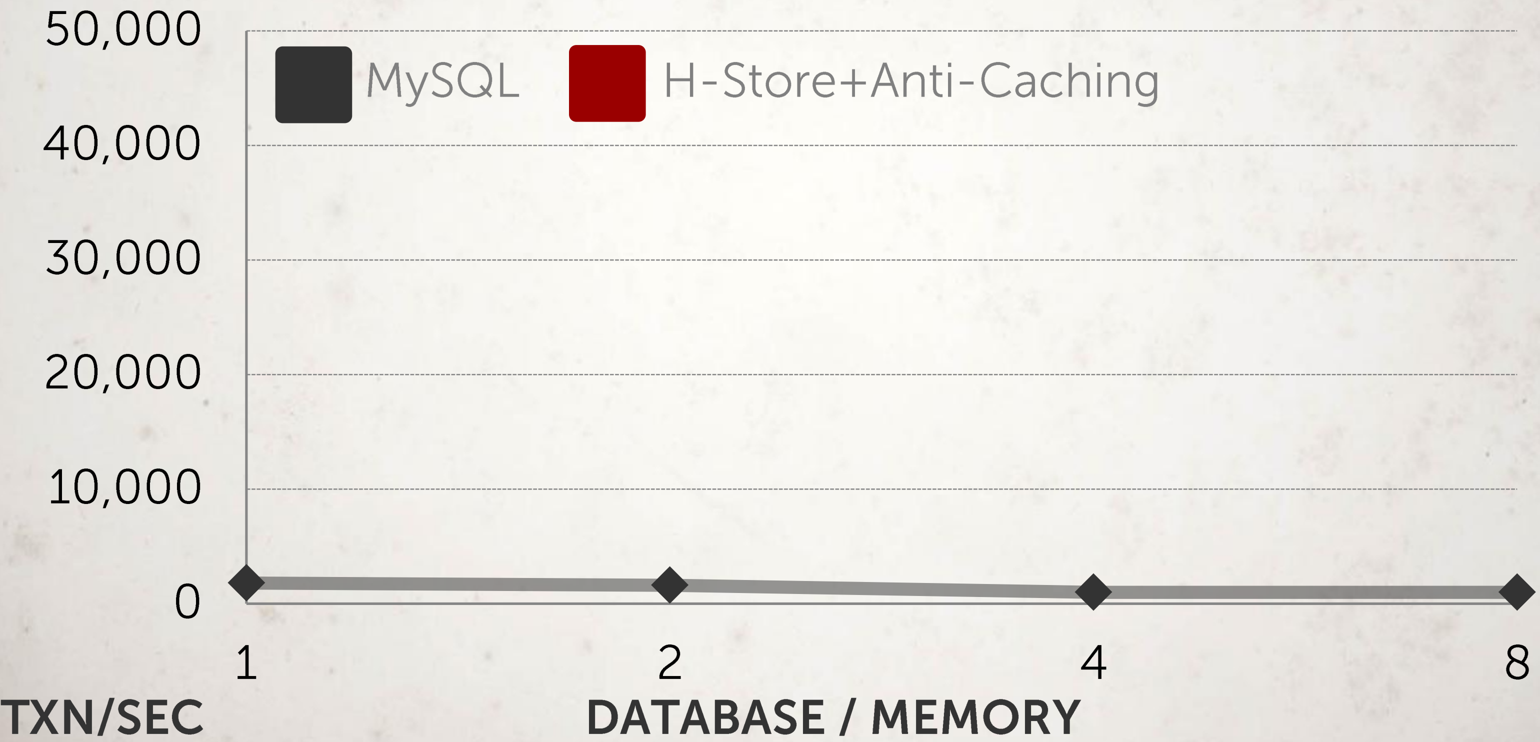

Application
Primary Storage
Anti-Cache


Application
Primary Storage
Anti-Cache


Application
Primary Storage
Anti-Cache

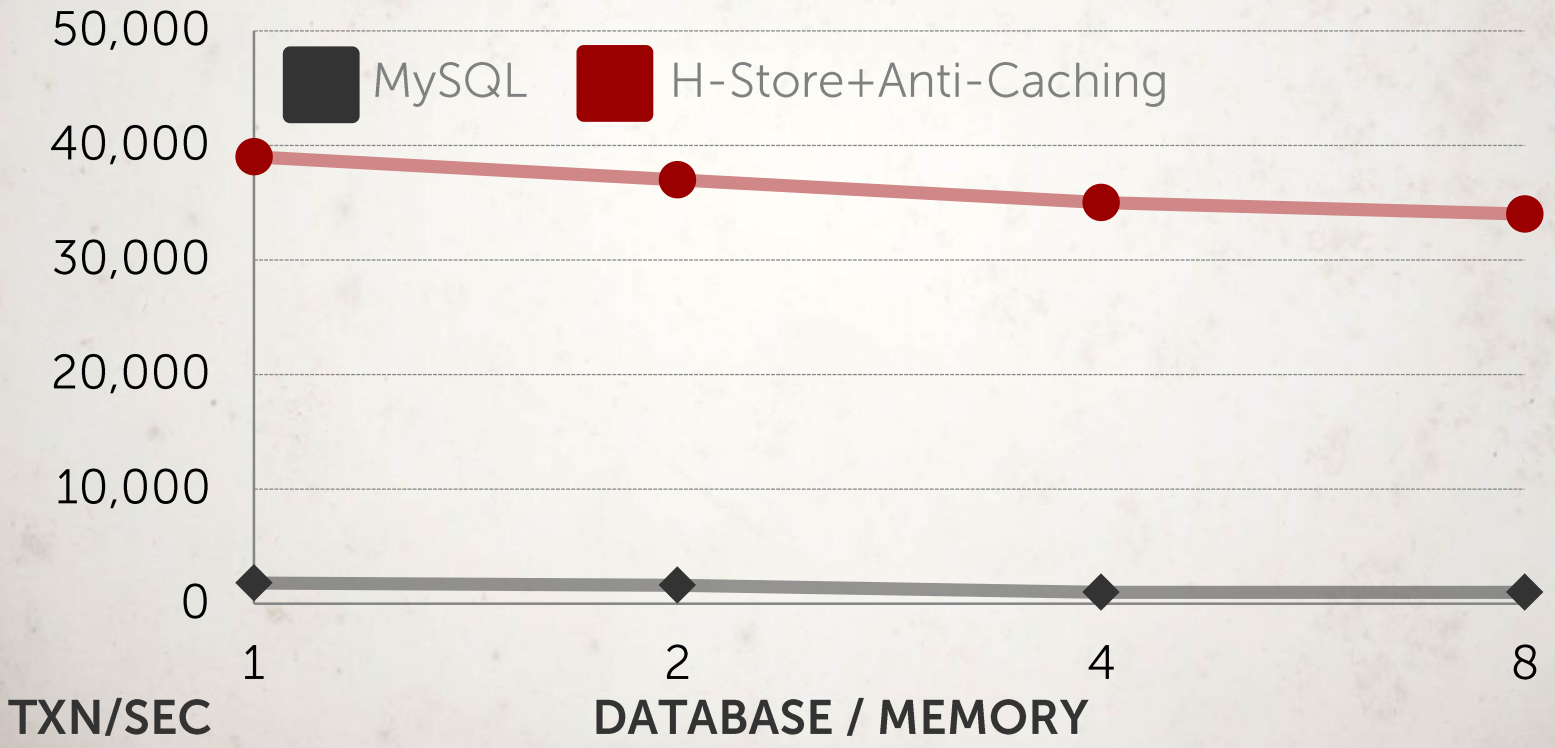
TPC-C Benchmark

100% Single-Partition Transactions – 100 Warehouses



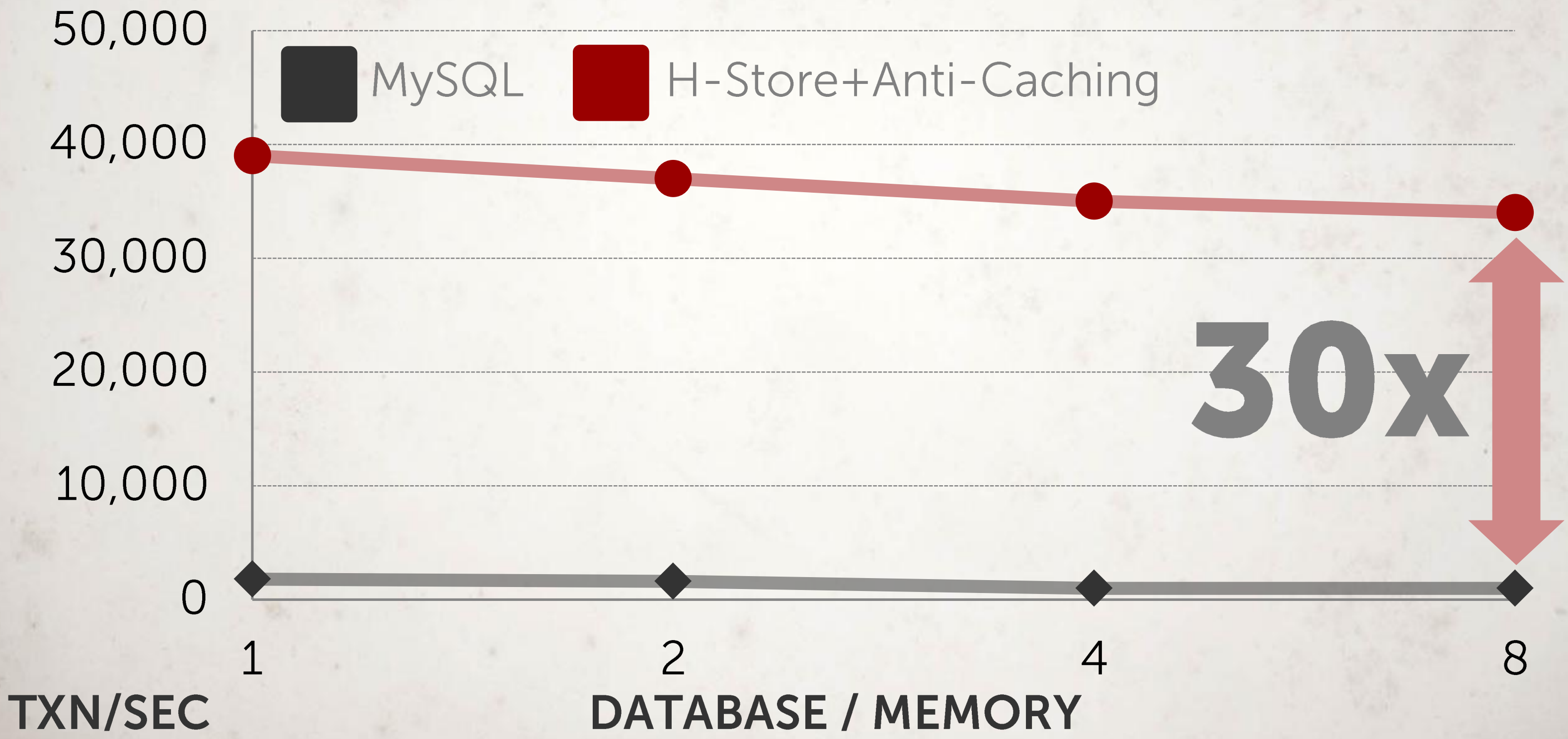
TPC-C Benchmark

100% Single-Partition Transactions – 100 Warehouses



TPC-C Benchmark

100% Single-Partition Transactions – 100 Warehouses



Future Research @ CMU

- Many-Core Concurrency
- Non-Volatile Memory
- Geo-replicated DBMS

Spänner

SpäXner

F2

BillHoweDB

BillH[redacted]veDB

OpauSQL™

Oracle, Please Acquire Us

OpauSQL™ – Design Principles

- Don't treat the DB as a black box.
- Machine learning to understand intra- & inter-txn dependencies.
- Introspection of integrity constraints.

OLTP Application Library

- Examine open source software to create a catalog of application properties.
- Automatically infer optimizations by examining DB access patterns.

Conclusion

- Most NewSQL systems are using known ideas to achieve high-performance for OLTP workloads.
- Database systems research is back at CMU.

END

@ANDY_PAVLO