Lock-Free Tries
Designs and Applications

Miguel Areias
joint work with Ricardo Rocha

CRACS & INESC-TEC LA
Faculty of Sciences, University of Porto, Portugal
miguel-areias@dcc.fc.up.pt *ricroc@dcc.fc.up.pt*

Topics in Discussion

- In this talk we will be discussing:
  - **YapTab**: A single-threaded tabling framework:
    - Key concepts about tabling.
    - Table space example.
    - Trie structure internals.

Topics in Discussion

- In this talk we will be discussing:
  - **YapTab**: A single-threaded tabling framework:
    - Key concepts about tabling.
    - Table space example.
    - Trie structure internals.
  - **YapTab-Mt**: A multi-threaded tabling framework:
    - Table space: No-Sharing, Subgoal-Sharing and Full-Sharing.
    - Trie structure:
      - Lock-Based: Standard, Global and Try-Locks.
      - **Lock-Free**: Tries and Hash Tries.
      - Performance analysis.

Topics in Discussion

- In this talk we will be discussing:
  - **YapTab**: A single-threaded tabling framework:
    - Key concepts about tabling.
    - Table space example.
    - Trie structure internals.
  - **YapTab-Mt**: A multi-threaded tabling framework:
    - Table space: No-Sharing, Subgoal-Sharing and Full-Sharing.
    - Trie structure:
      - Lock-Based: Standard, Global and Try-Locks.
      - **Lock-Free**: Tries and Hash Tries.
      - Performance analysis.
  - **Lock-Free Tries** - Applications:
    - Asynchronous parallelism.
    - **Parallelization techniques**: Top-Down and Bottom-Up.
    - Performance analysis.

Tabling in Prolog Systems

- **Tabling** or **memoing** is an implementation technique that overcomes some of the limitations of Prolog resolution:
  - Tabled subgoals are evaluated by storing their answers in an appropriate data space, called the **table space**.
  - Repeated calls to tabled subgoals are resolved by **consuming** the answers already stored in the table instead of **being re-evaluated** against the program clauses.

Tabling in Prolog Systems

- **Tabling** or **memoing** is an implementation technique that overcomes some of the limitations of Prolog resolution:
  - Tabled subgoals are evaluated by storing their answers in an appropriate data space, called the **table space**.
  - Repeated calls to tabled subgoals are resolved by **consuming** the answers already stored in the table instead of **being re-evaluated** against the program clauses.

- Implementations of **Tabling** are currently available in systems like:
  - XSB Prolog, **Yap Prolog**, B-Prolog, ALS-Prolog, Mercury, Ciao Prolog.

Tabling in Prolog Systems

- **Tabling** or **memoing** is an implementation technique that overcomes some of the limitations of Prolog resolution:
  - Tabled subgoals are evaluated by storing their answers in an appropriate data space, called the **table space**.
  - Repeated calls to tabled subgoals are resolved by **consuming** the answers already stored in the table instead of **being re-evaluated** against the program clauses.

- Implementations of **Tabling** are currently available in systems like:
  - XSB Prolog, **Yap Prolog**, B-Prolog, ALS-Prolog, Mercury, Ciao Prolog.

- **Multithreading** combined with **Tabling**:
  - XSB Prolog.
  - **Yap Prolog (YapTab-Mt)**.

Table Space - Example

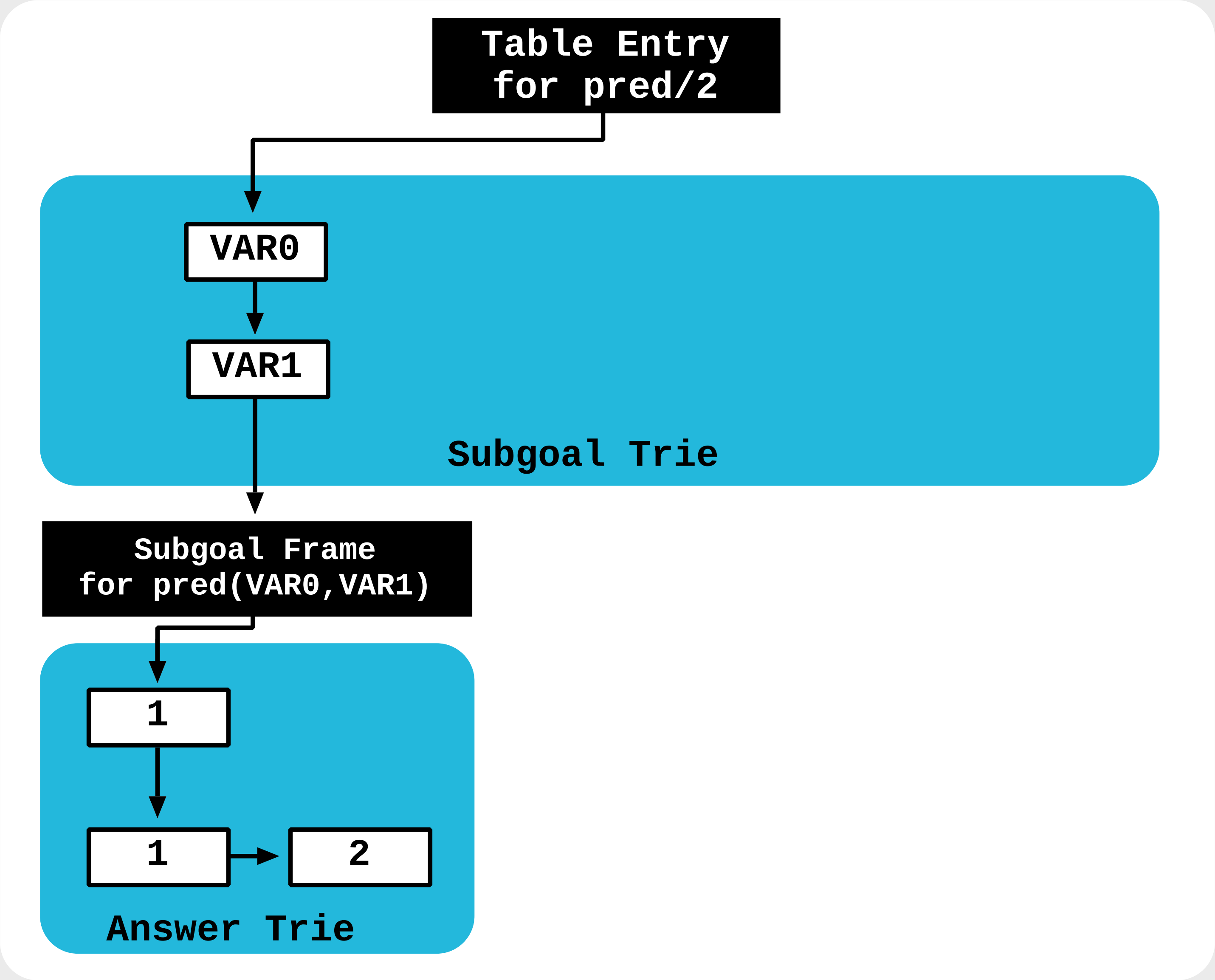


Table Space - Example

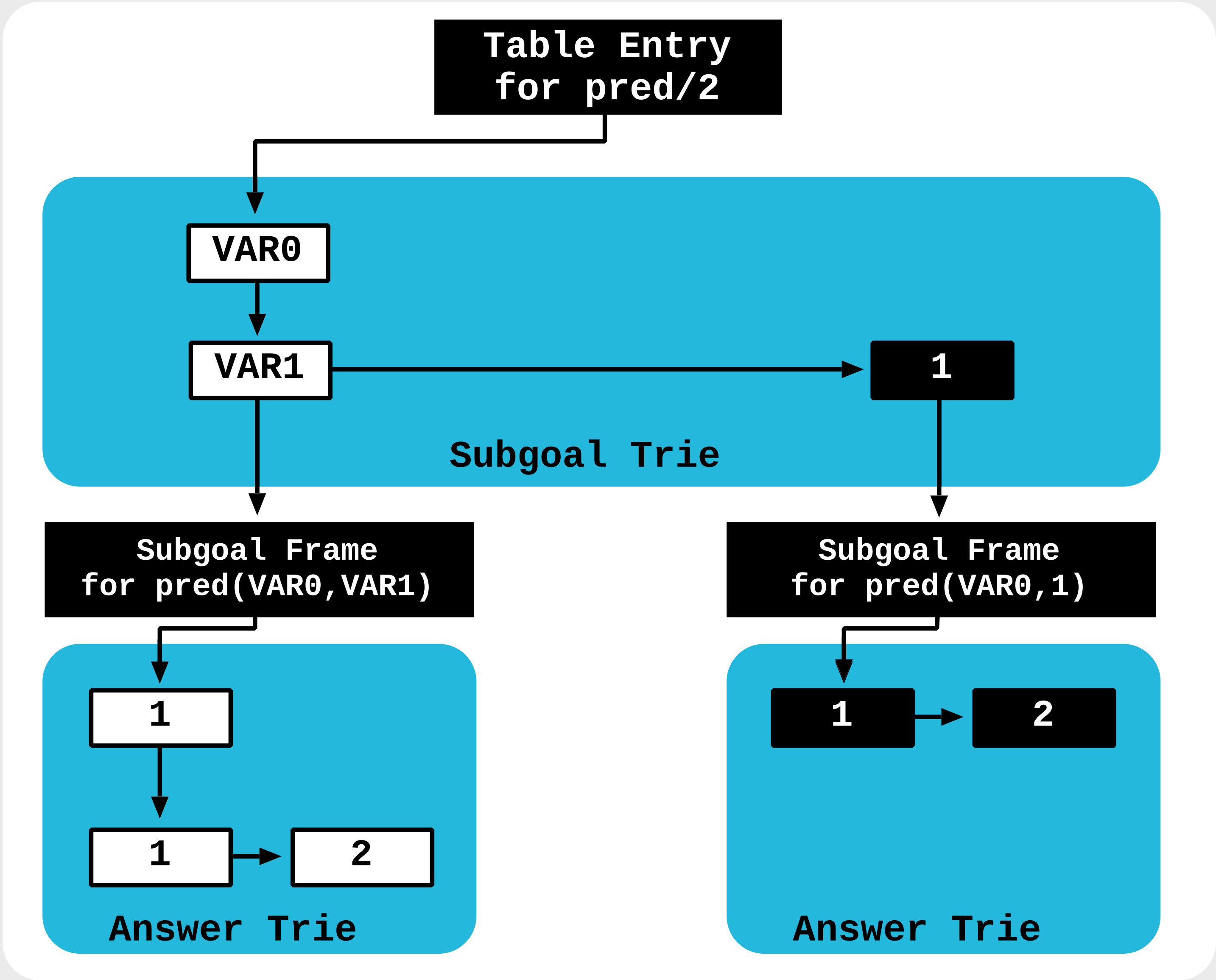


Table Space - Trie Level Internals

- All trie levels have one **parent (P)** node and at least one **child (K)** node.
- Only **search** and **insert** operations are executed on the trie levels.
- **Insertion** of new nodes is done on the **head of the chain**, until a **threshold** is achieved.

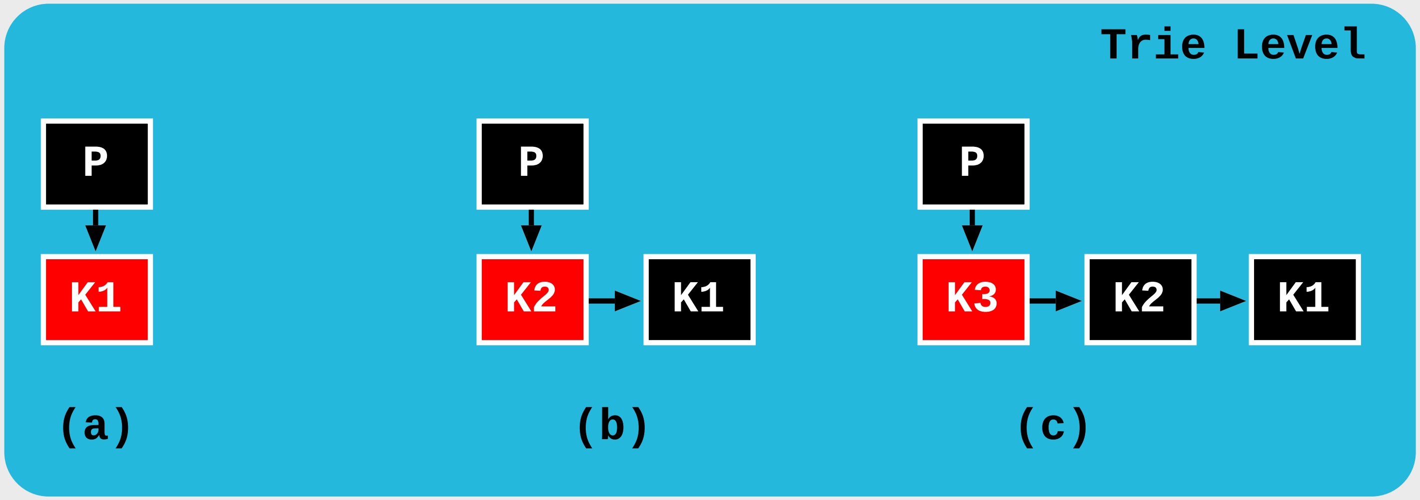


Table Space - Trie Level Internals

- When the **threshold** is achieved, a **hashing mechanism with separating chaining** is added to the level.
- The **hash H** node stores generic information about the level.
- The **value B** is the number of bucket entries.
- When the hash becomes **saturated**, it is **expanded** to a new hash with $2 * B$ bucket entries.

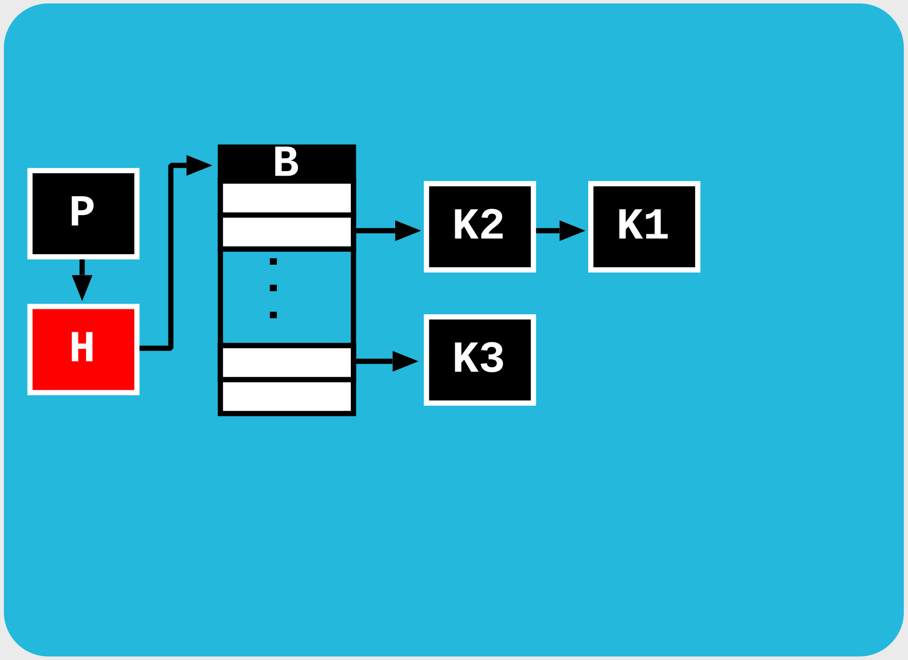


Table Space - Trie Level Internals

- When the **threshold** is achieved, a **hashing mechanism with separating chaining** is added to the level.
- The **hash H** node stores generic information about the level.
- The **value B** is the number of bucket entries.
- When the hash becomes **saturated**, it is **expanded** to a new hash with $2 * B$ bucket entries.

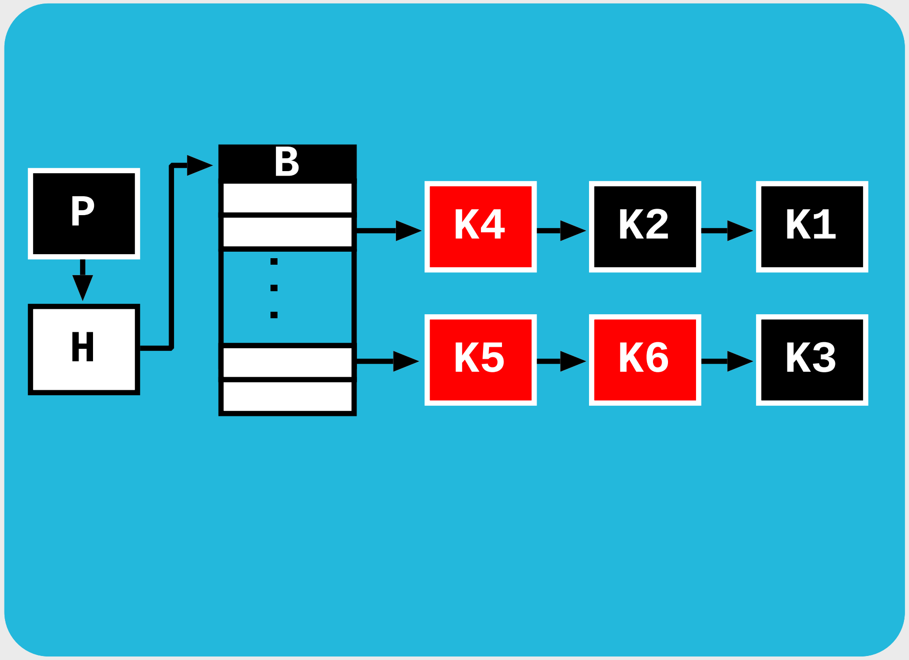


Table Space - Trie Level Internals

- When the **threshold** is achieved, a **hashing mechanism with separating chaining** is added to the level.
- The **hash H** node stores generic information about the level.
- The **value B** is the number of bucket entries.
- When the hash becomes **saturated**, it is **expanded** to a new hash with $2 * B$ bucket entries.

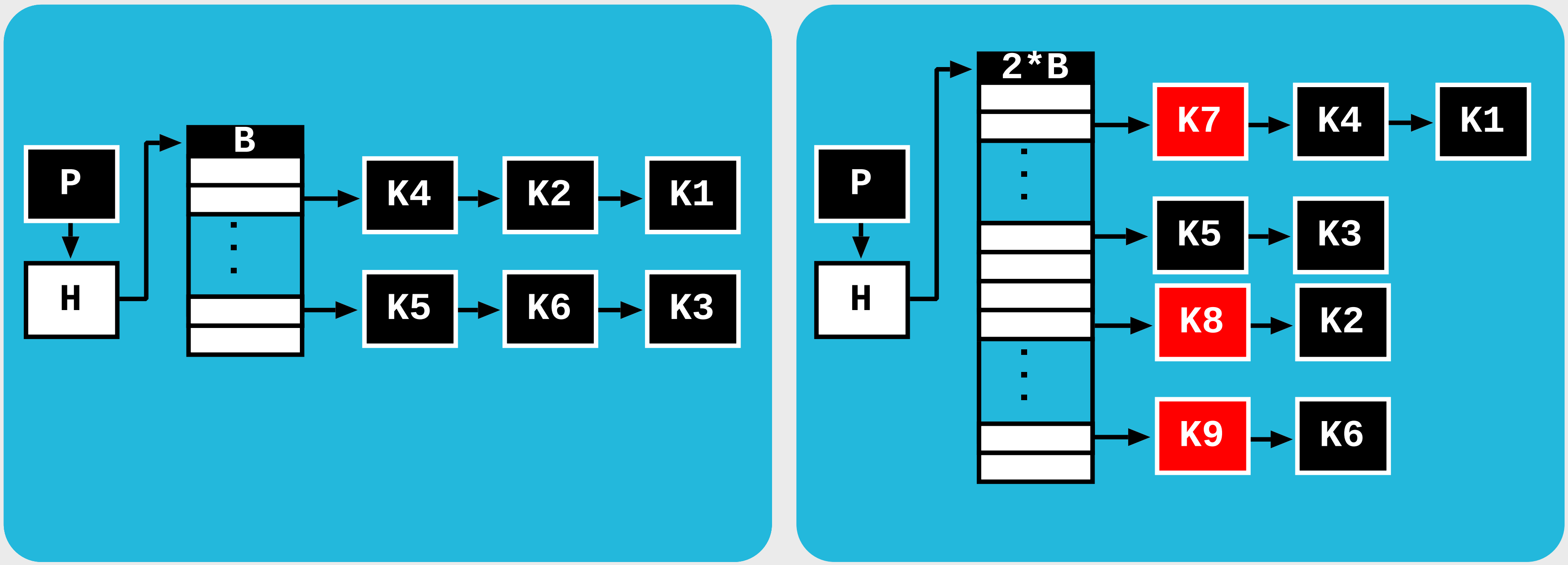


Table Space - Multithreaded Designs

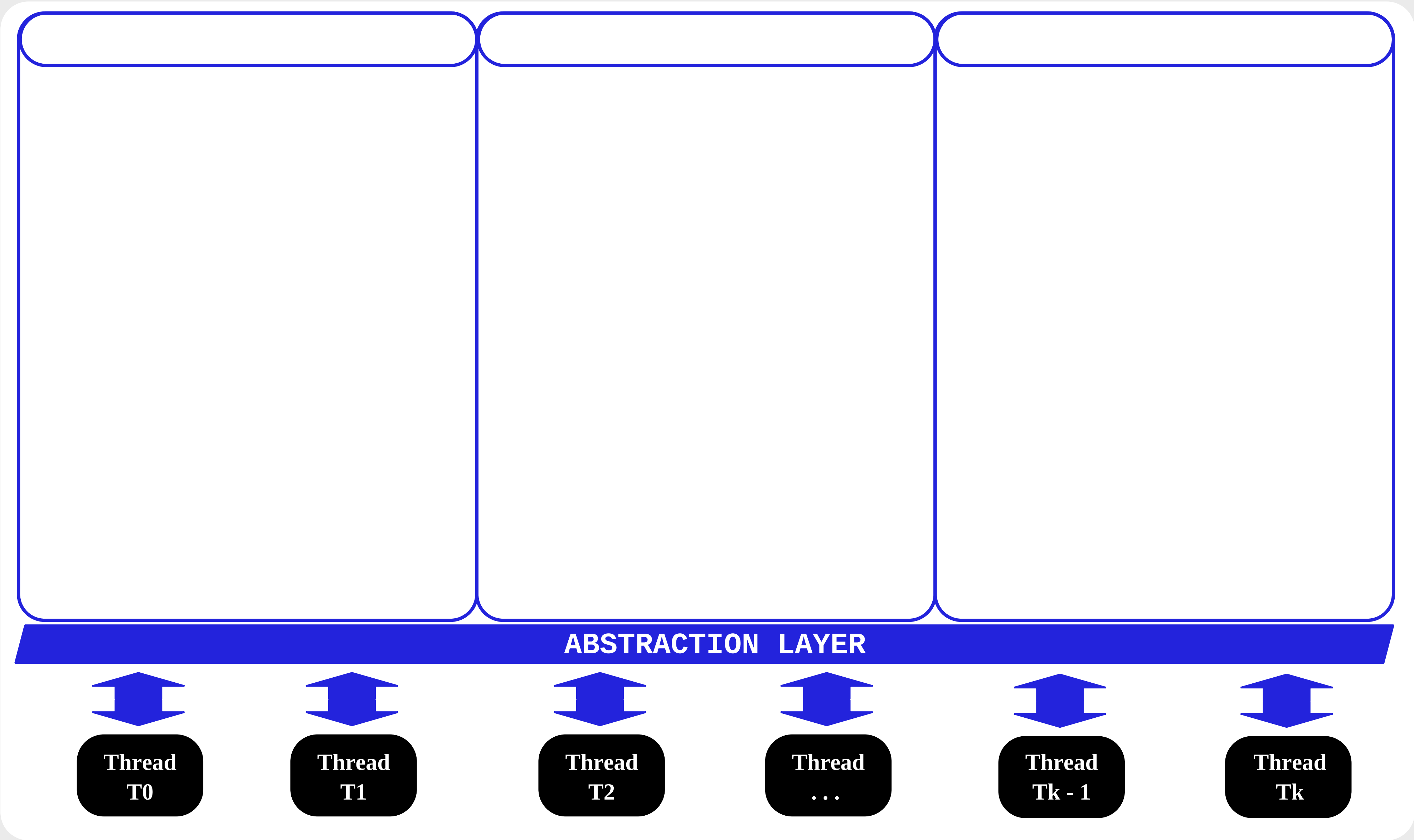


Table Space - Multithreaded Designs

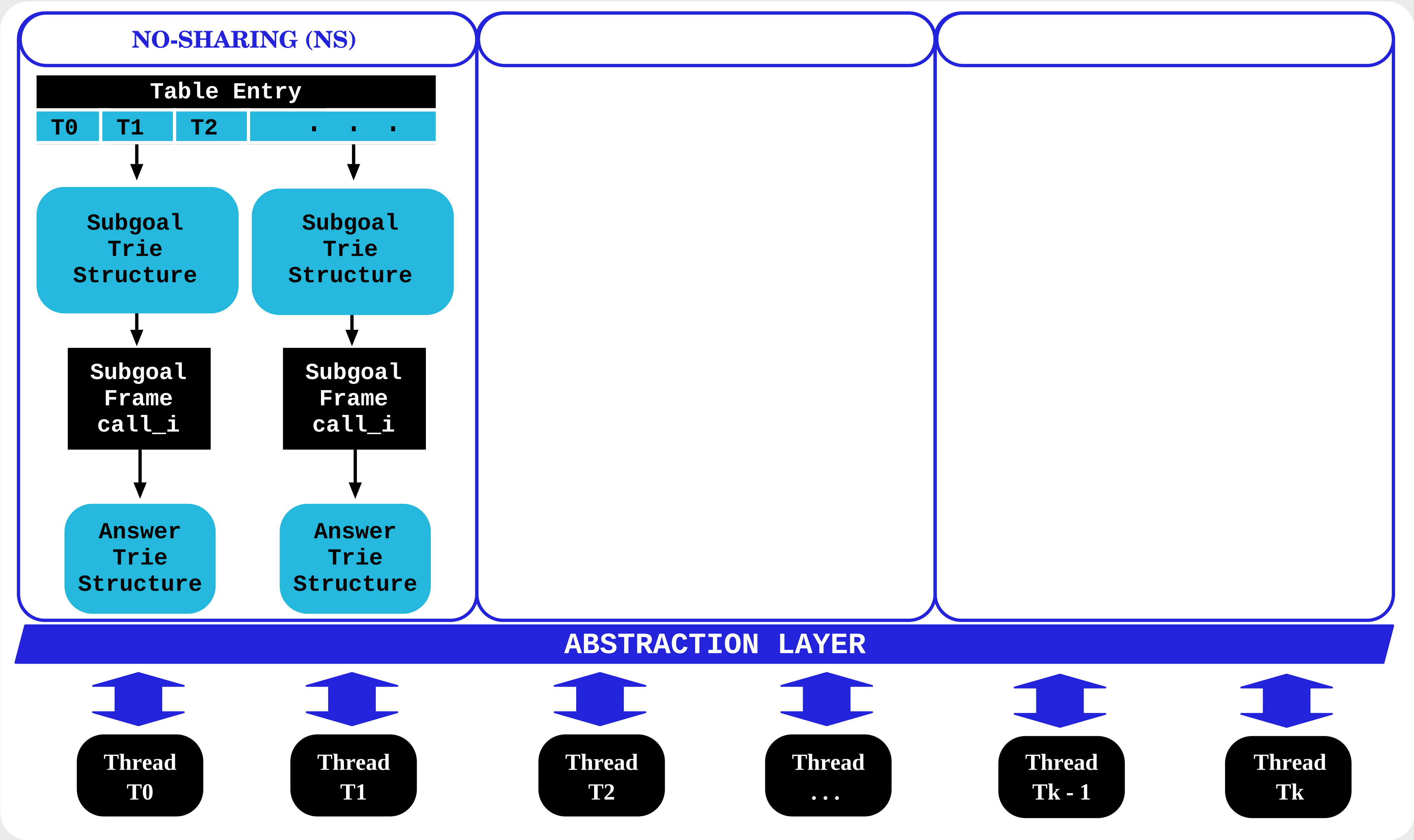


Table Space - Multithreaded Designs

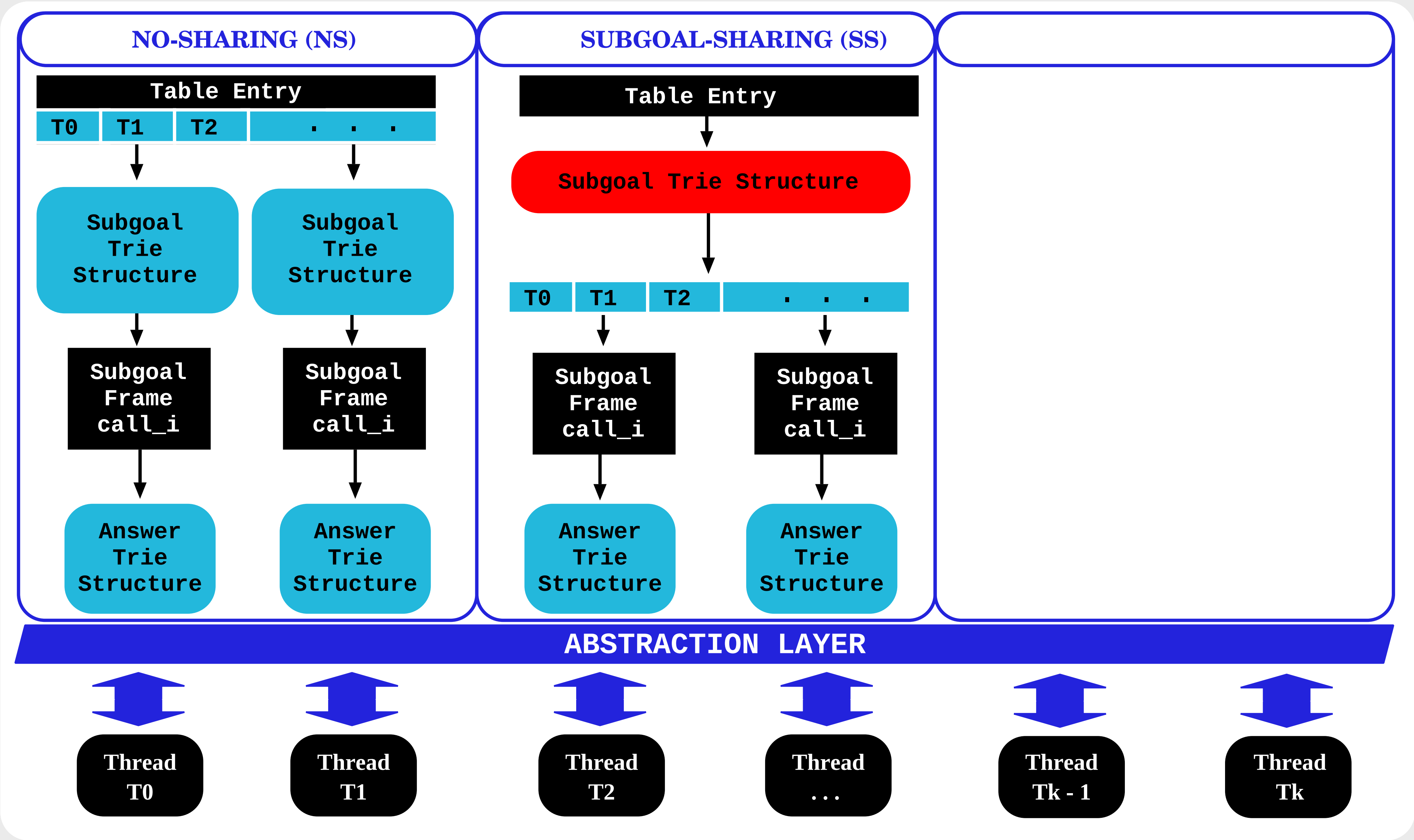
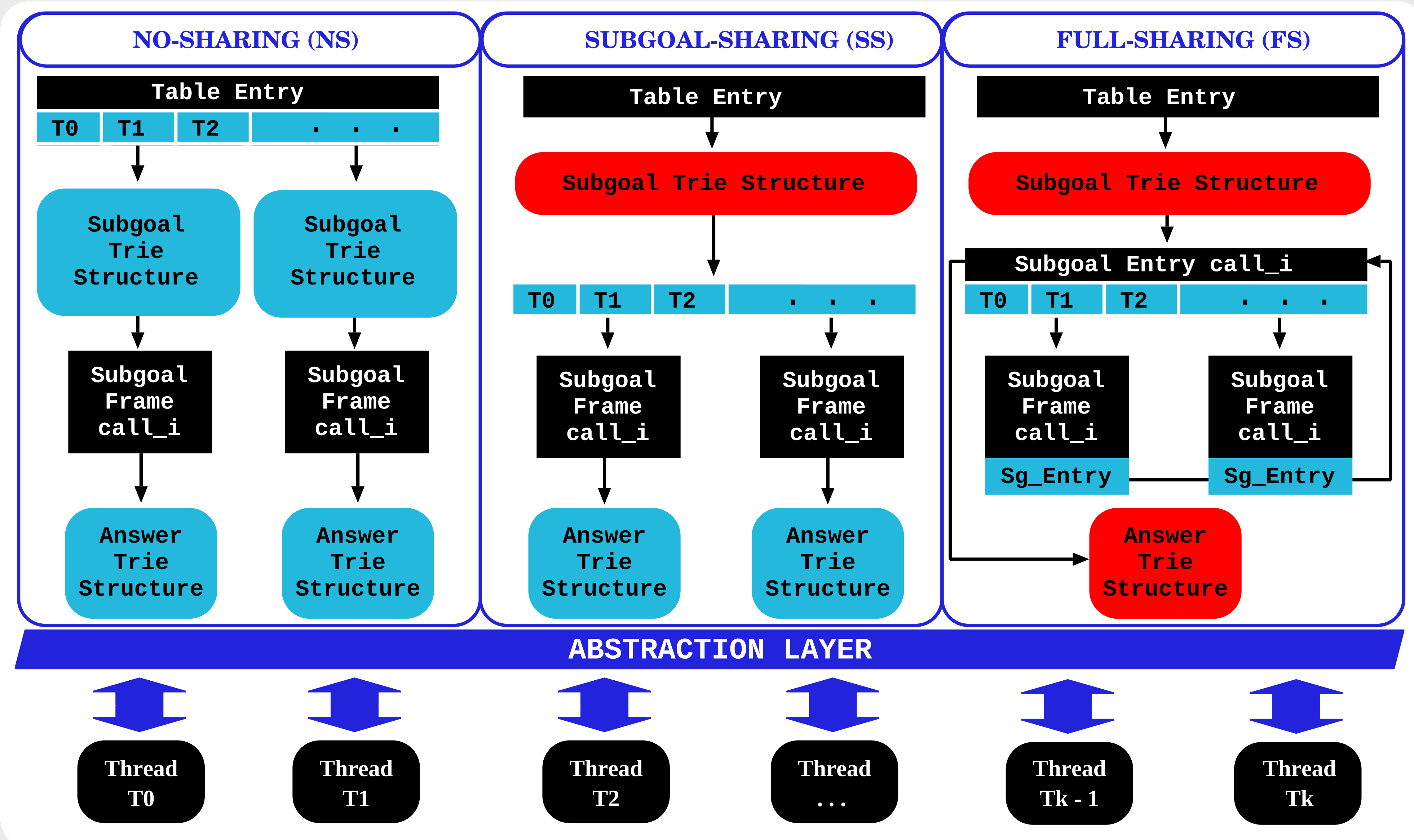


Table Space - Multithreaded Designs



Lock-Free Tries - Motivation

- **Until now** to deal with concurrency we used **locks**:
  - Lock Type:
    - Standard Locks.
    - Try-Locks.
  - Lock Location:
    - Field per trie node.
    - Global array of lock entries.
- The **expansion** of the hash **locked** the **insertion** and could in some cases **delay** the **search** operation (inefficiency).

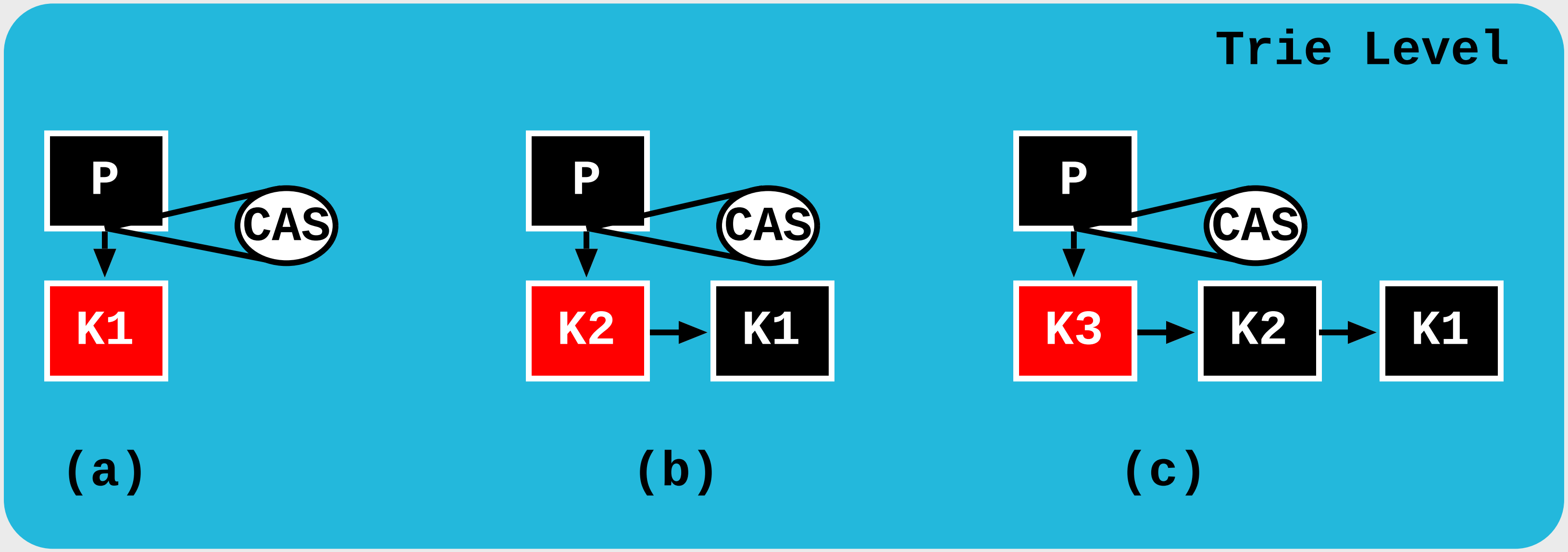
Lock-Free Tries - Motivation

- **Until now** to deal with concurrency we used **locks**:
  - Lock Type:
    - Standard Locks.
    - Try-Locks.
  - Lock Location:
    - Field per trie node.
    - Global array of lock entries.
- The **expansion** of the hash **locked** the **insertion** and could in some cases **delay** the **search** operation (inefficiency).
- **With lock-free** we are interested in **reducing** the granularity of the **synchronization**, by taking advantage of the **CAS (Compare-and-Swap)** operation.
  - Nowadays **can be found** on many of the **common architectures**.
  - At the **heart** of many **lock-free objects**.

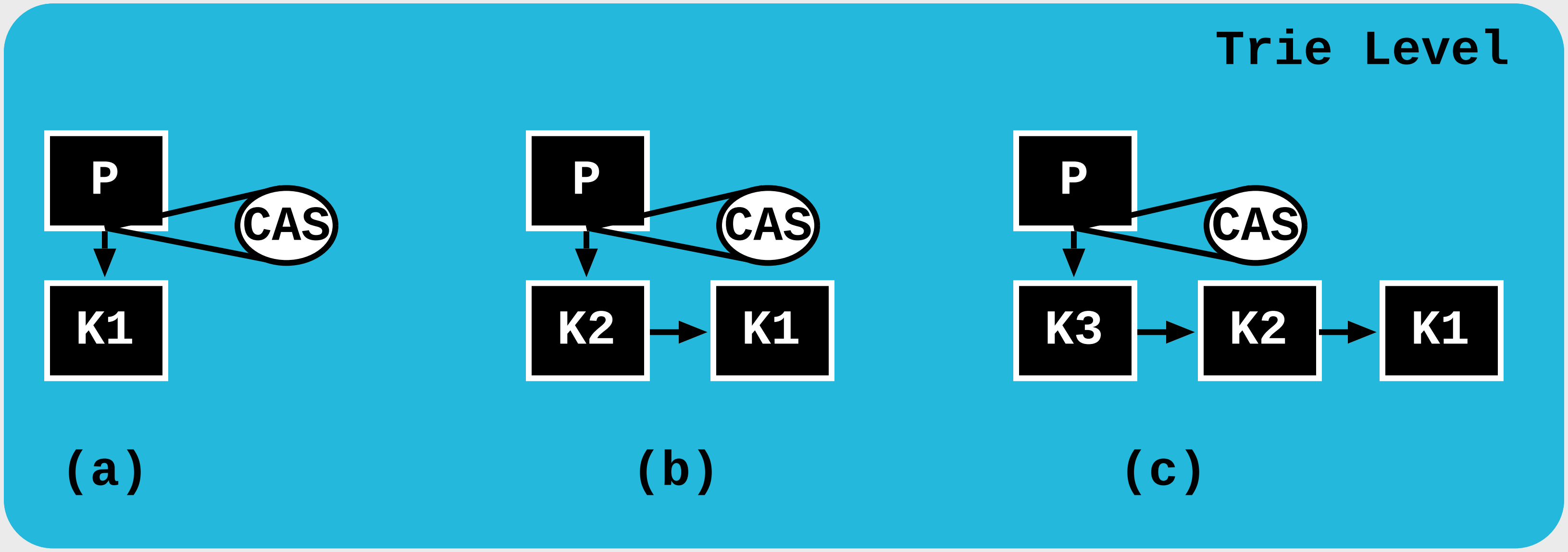
Lock-Free Tries - Motivation

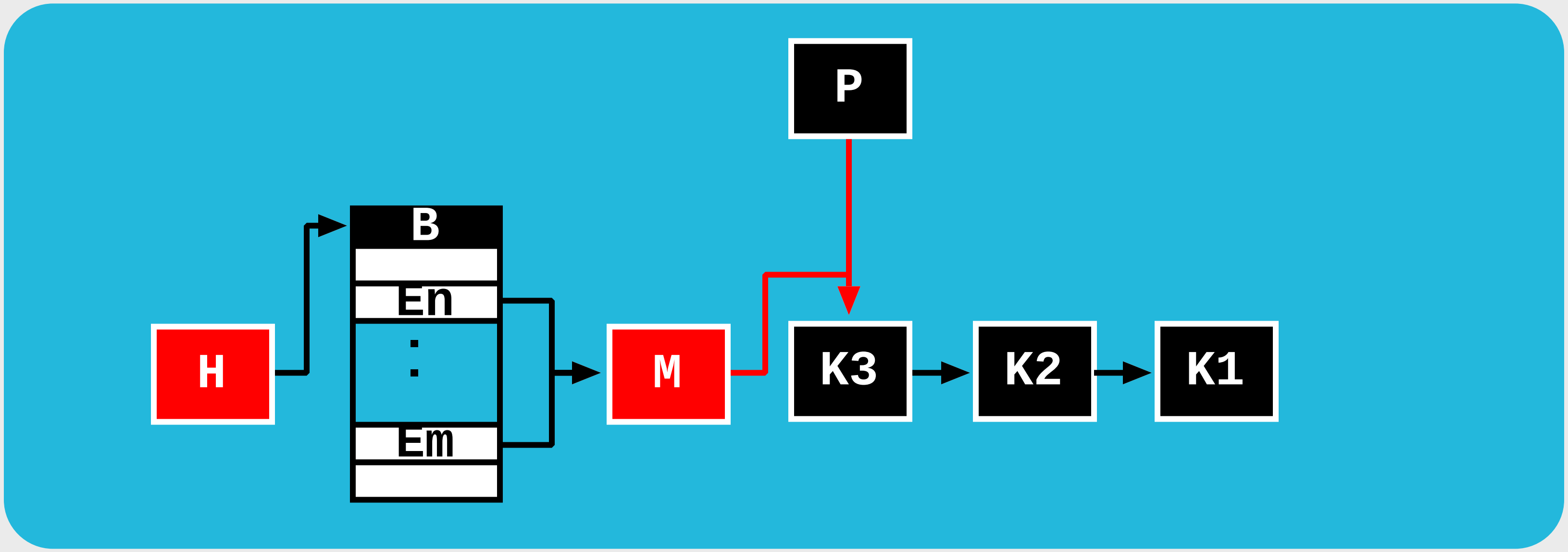
- **Lock-free linearizable objects** permit a **greater concurrency** since **semantically consistent** (non-interfering) operations **may execute in parallel**.
- Several **lock-free models** do exist:
  - Shalev and Shavit **Split-Ordered Lists**
  - Prokopec **Concurrent Tries**
  - Cliff's **Non-Blocking Hash Tables**.
- But ... **none** of the existent models is specifically **aimed** for an environment with the **characteristics** of our **tabling framework**.
  - Support for the **concurrent deletion** of nodes **increases** the **complexity** of the models.

Lock-Free Tries - The First Expansion

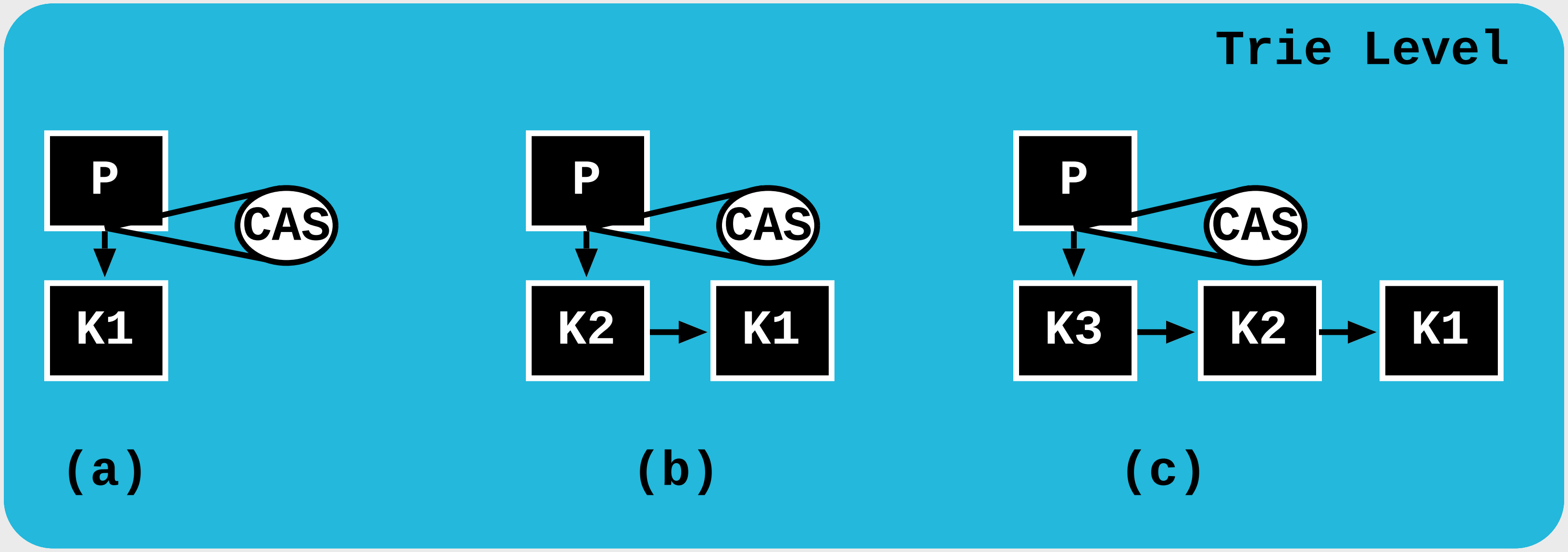


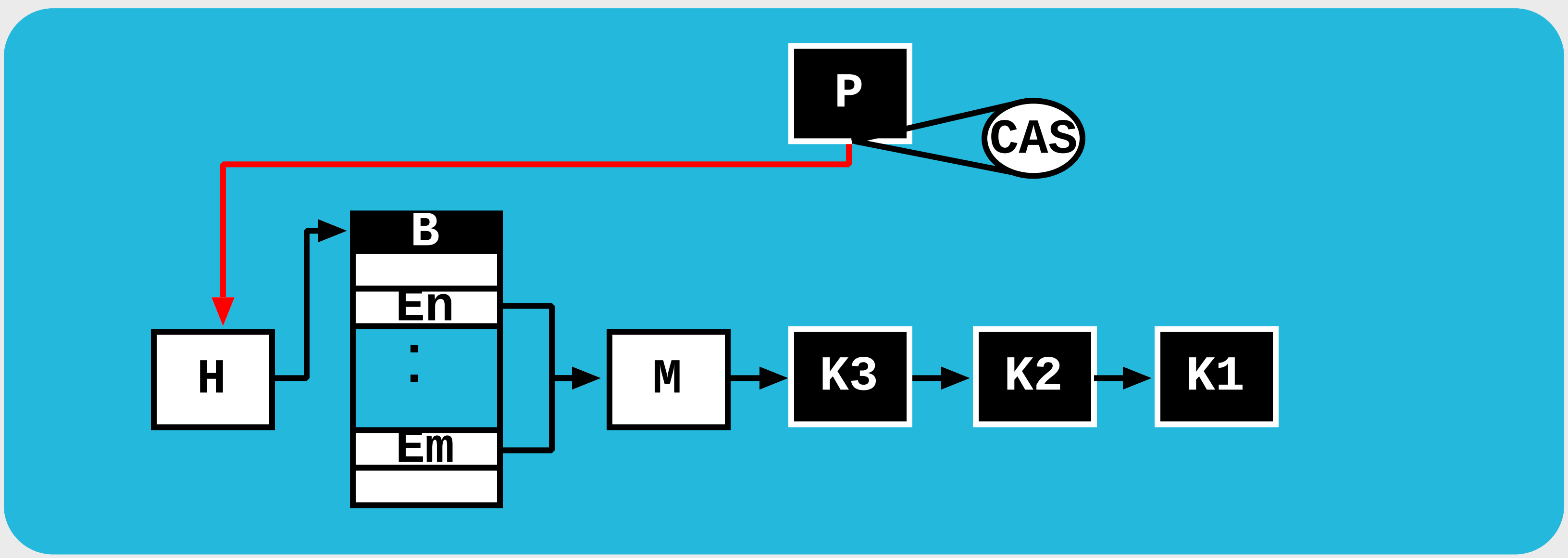
Lock-Free Tries - The First Expansion



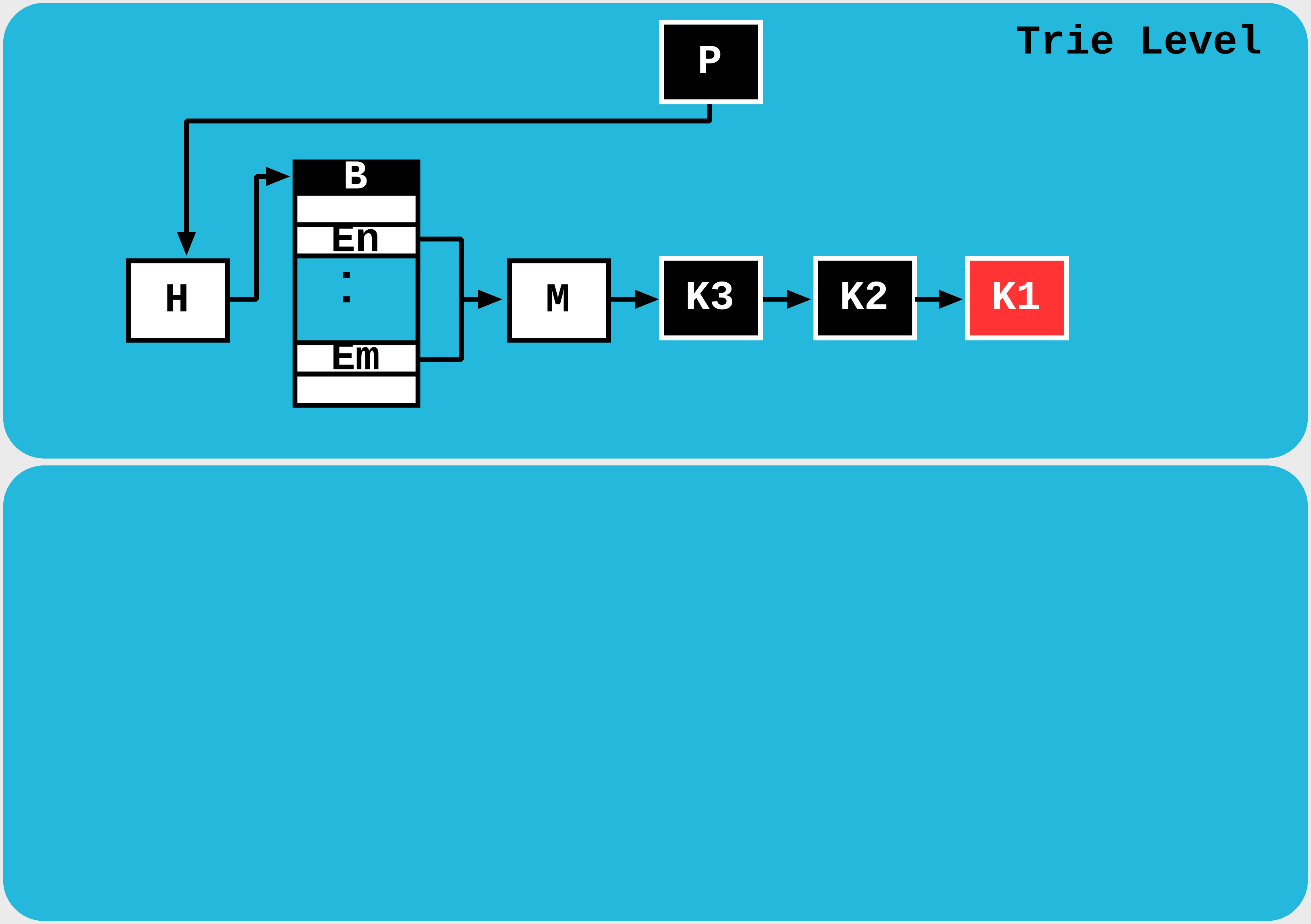


Lock-Free Tries - The First Expansion

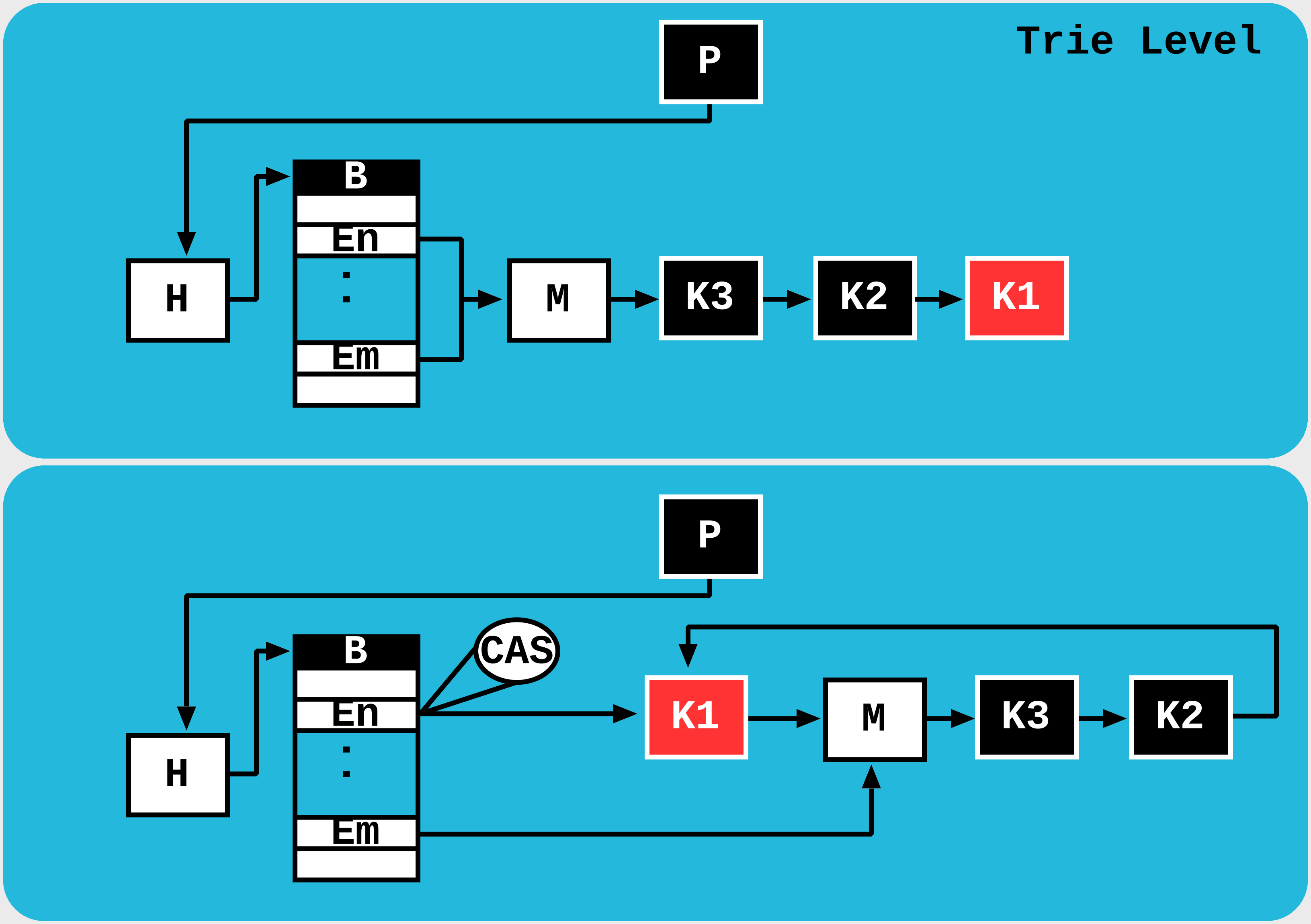




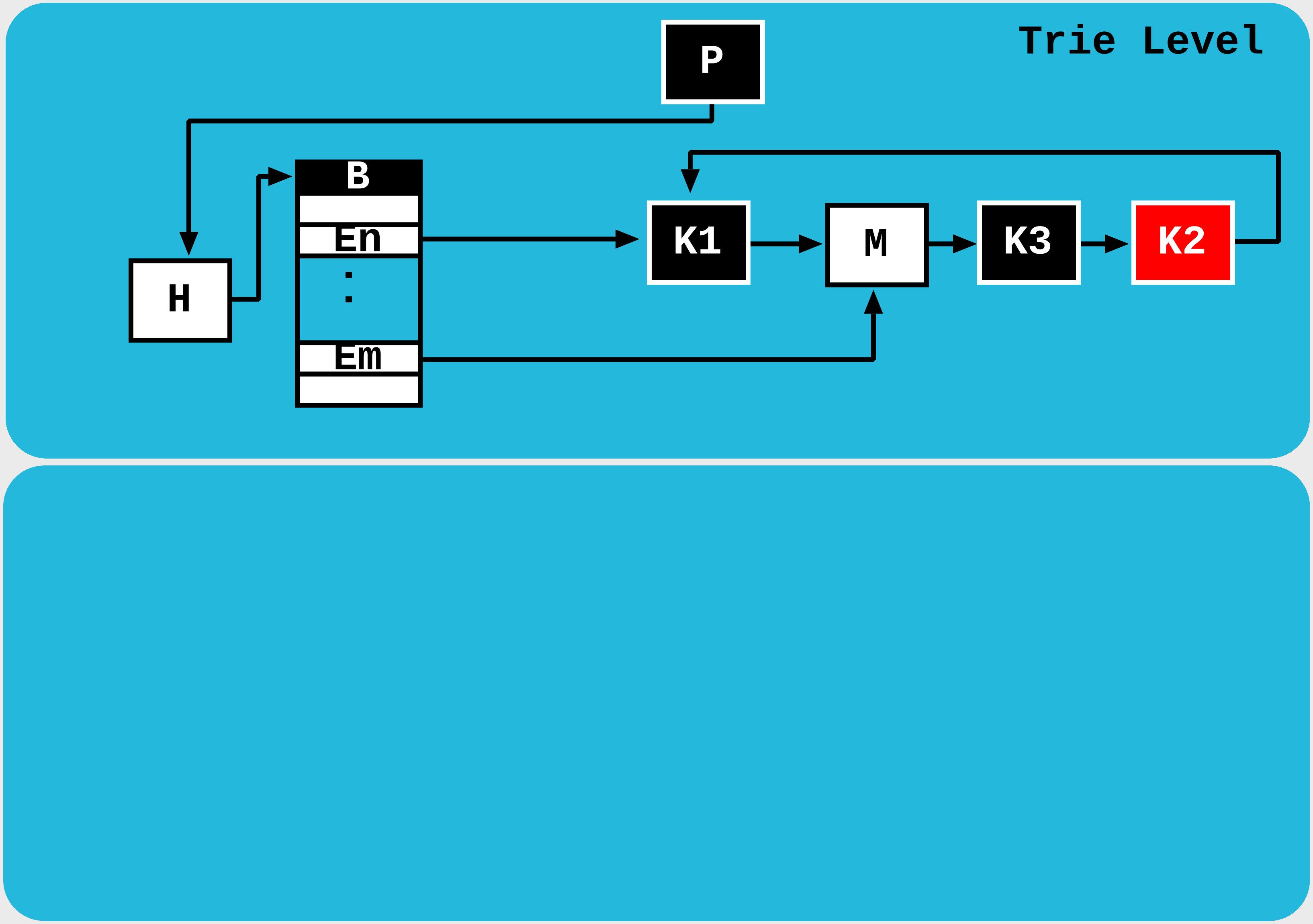
Lock-Free Tries - The First Expansion



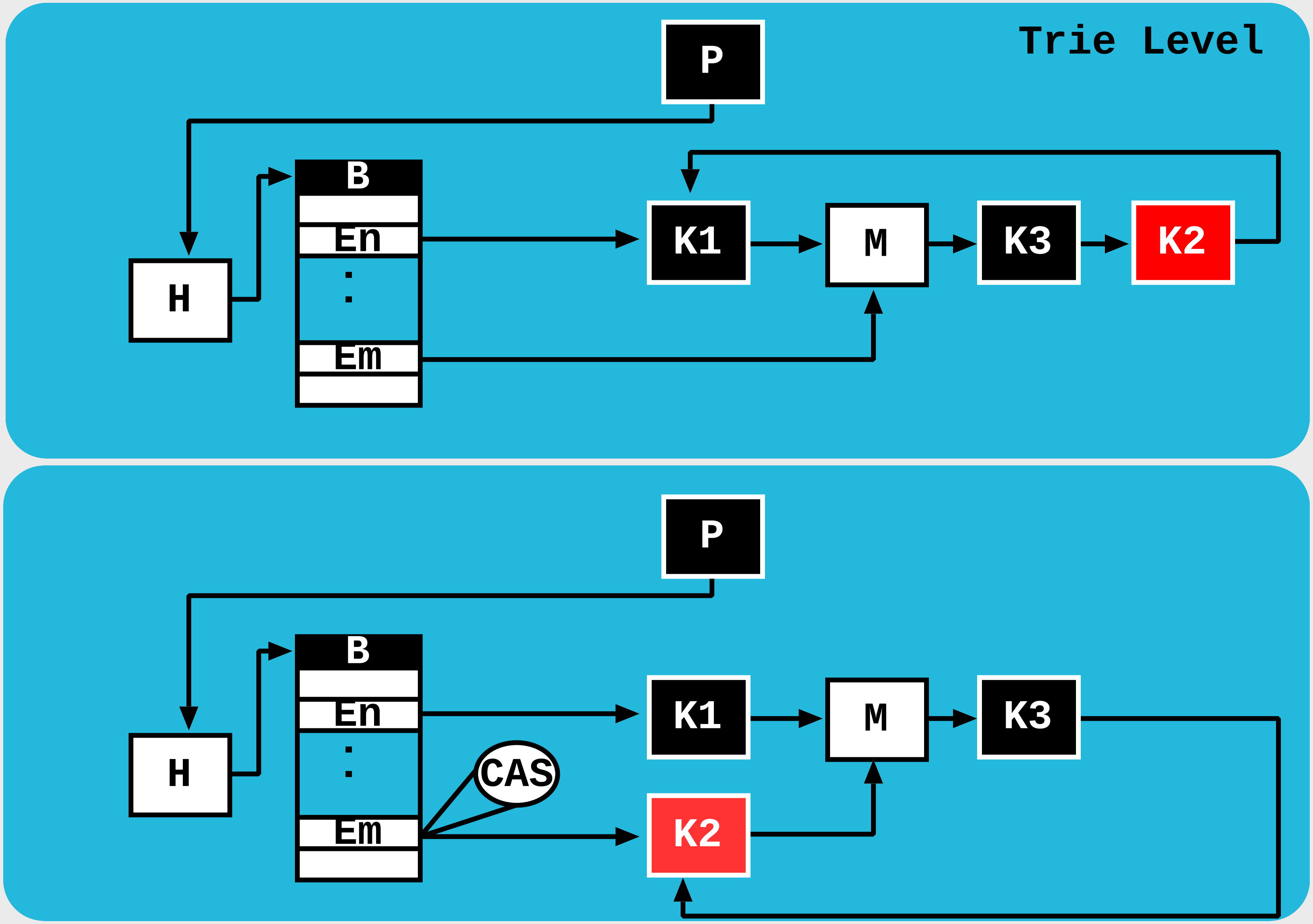
Lock-Free Tries - The First Expansion



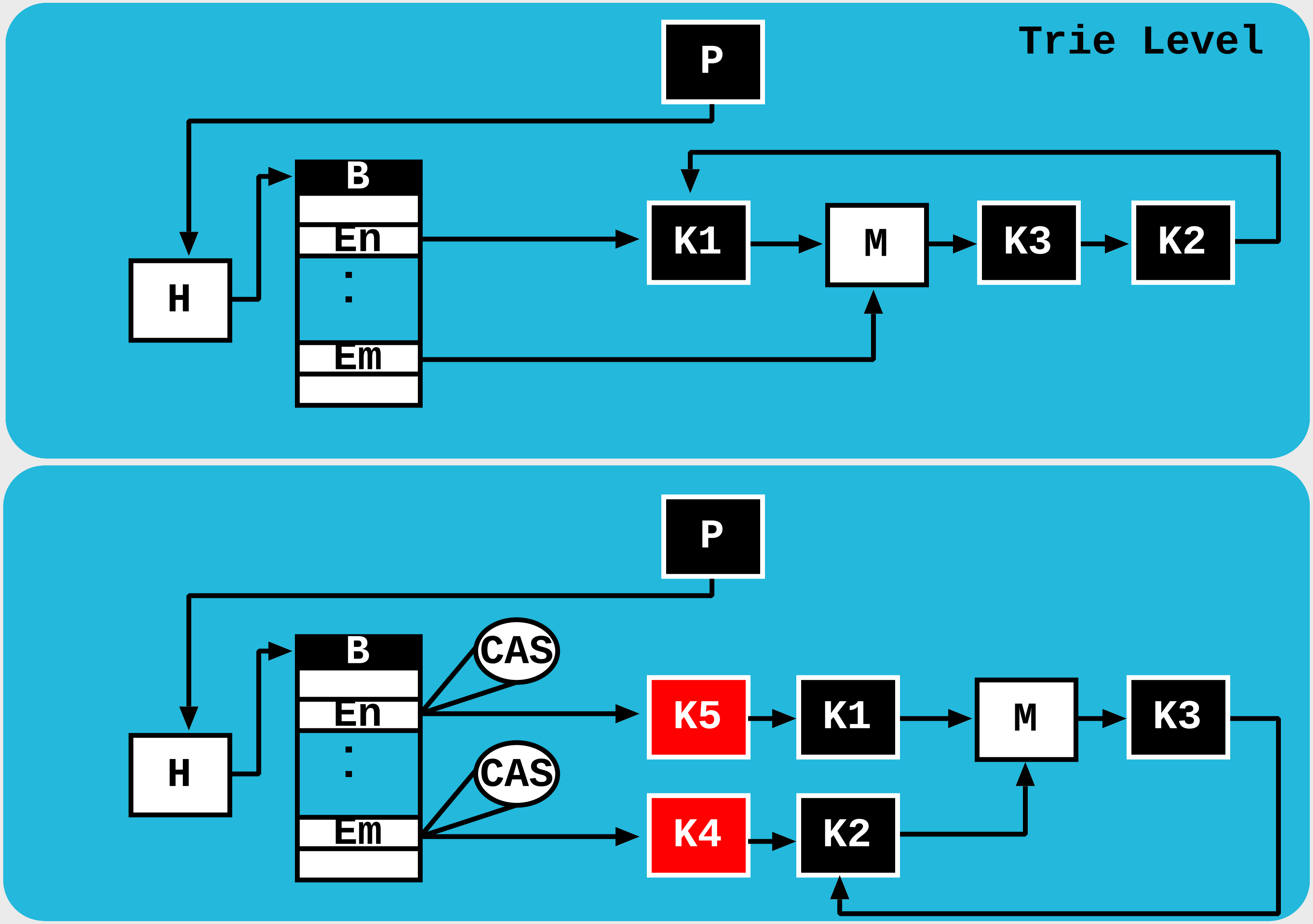
Lock-Free Tries - The First Expansion



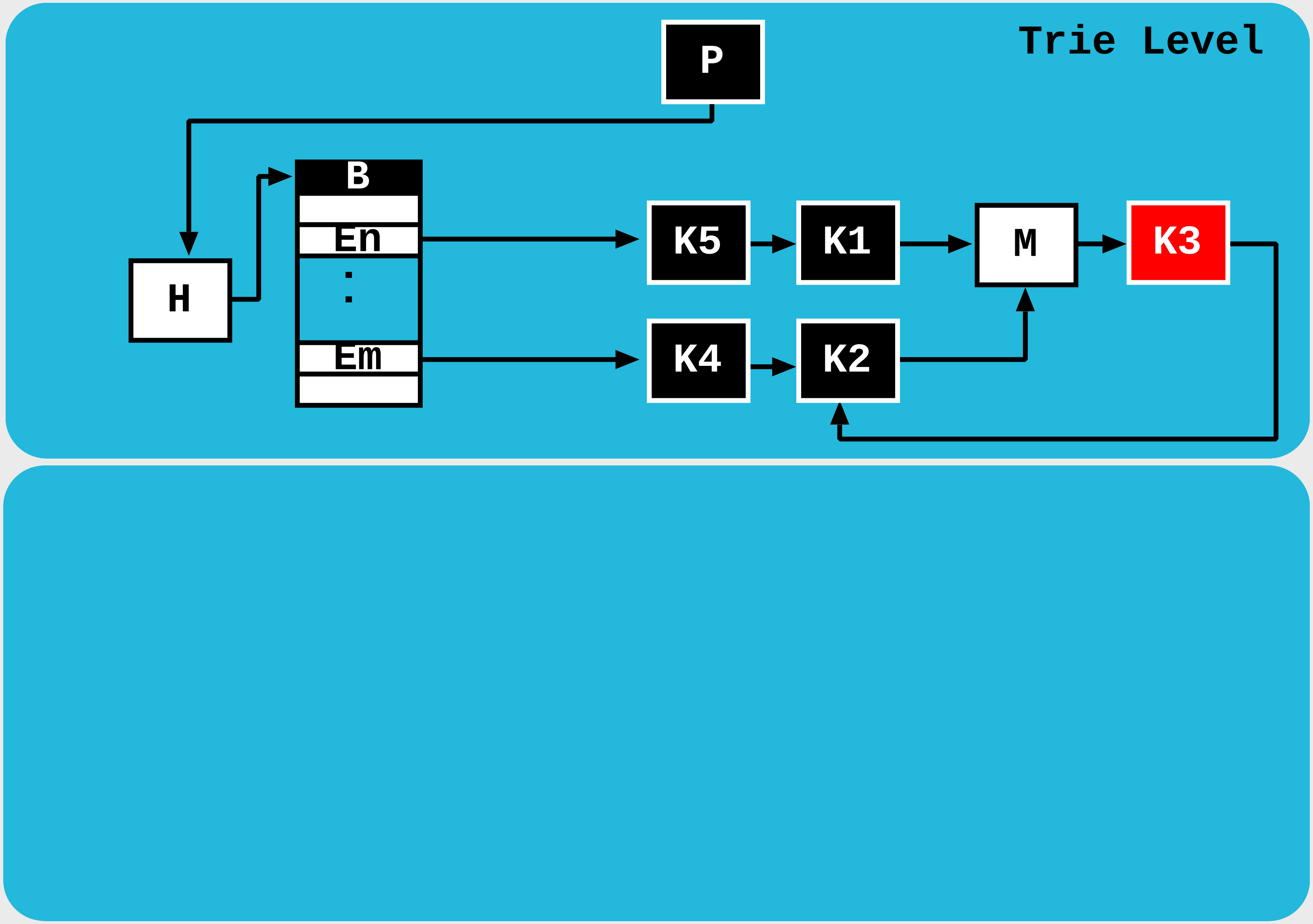
Lock-Free Tries - The First Expansion



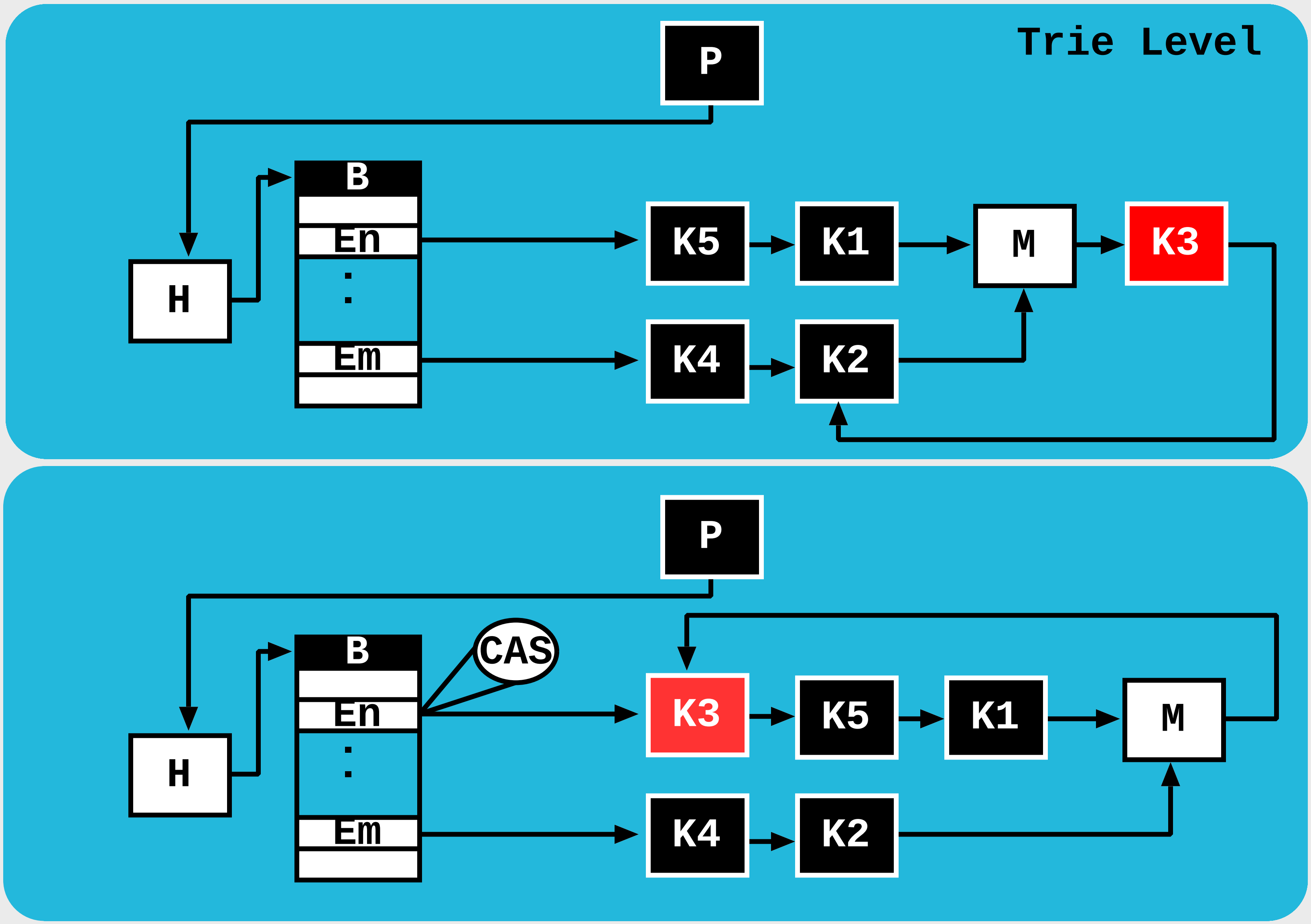
Lock-Free Tries - The First Expansion



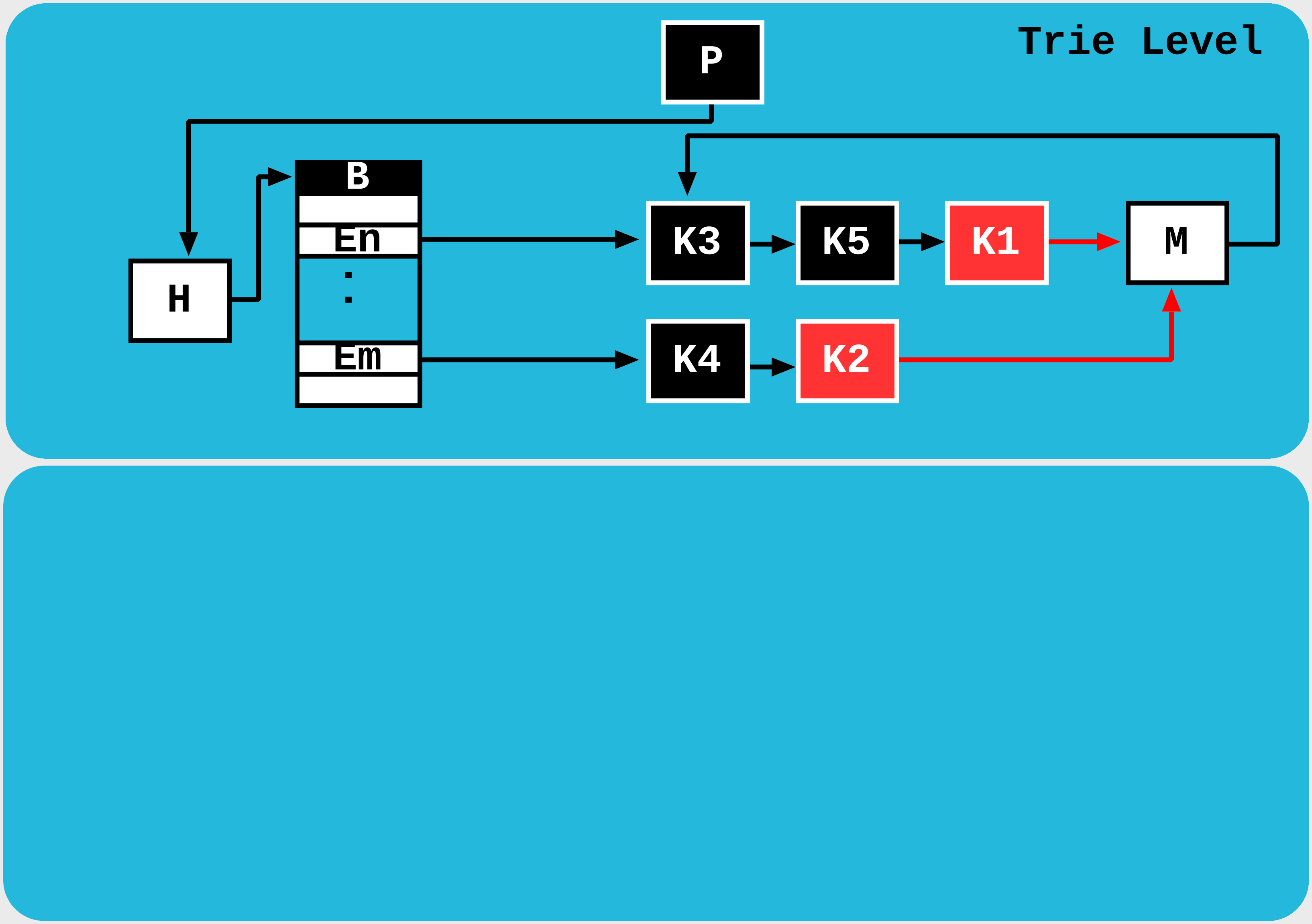
Lock-Free Tries - The First Expansion



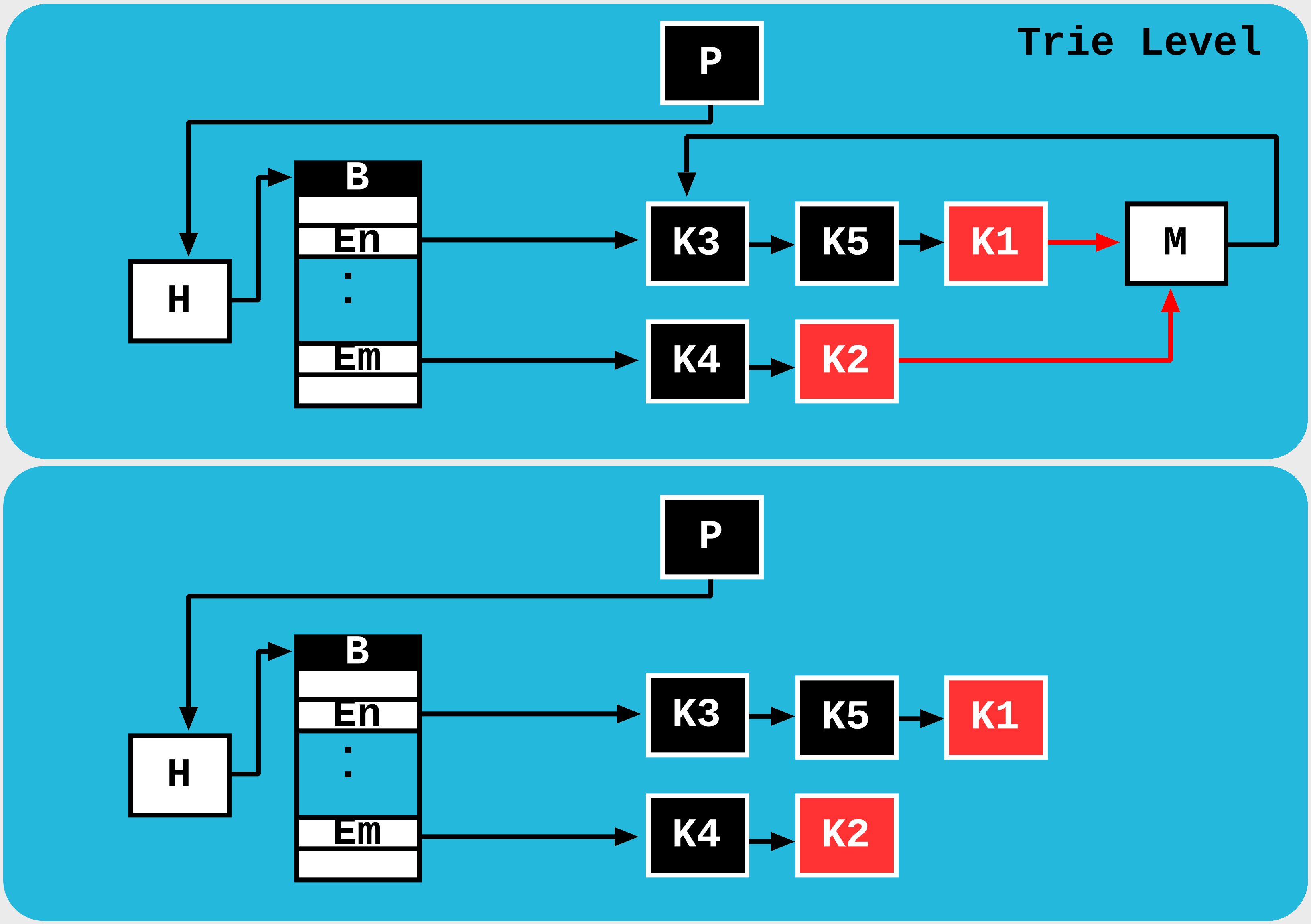
Lock-Free Tries - The First Expansion



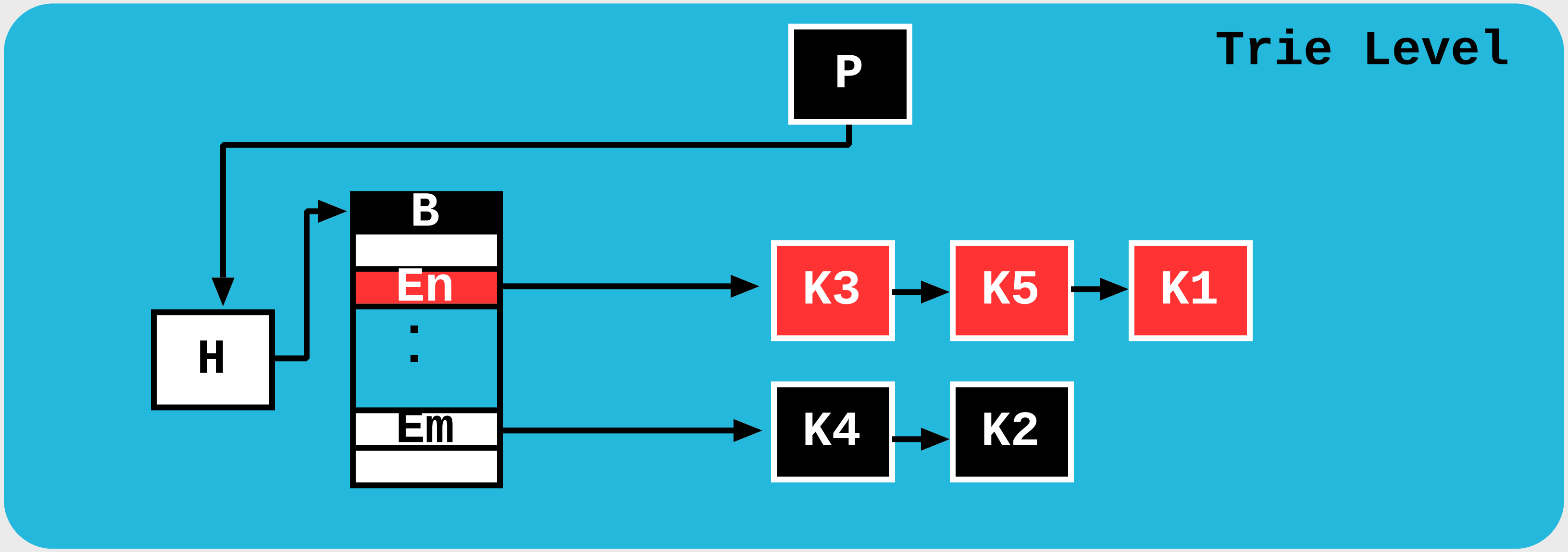
Lock-Free Tries - The First Expansion



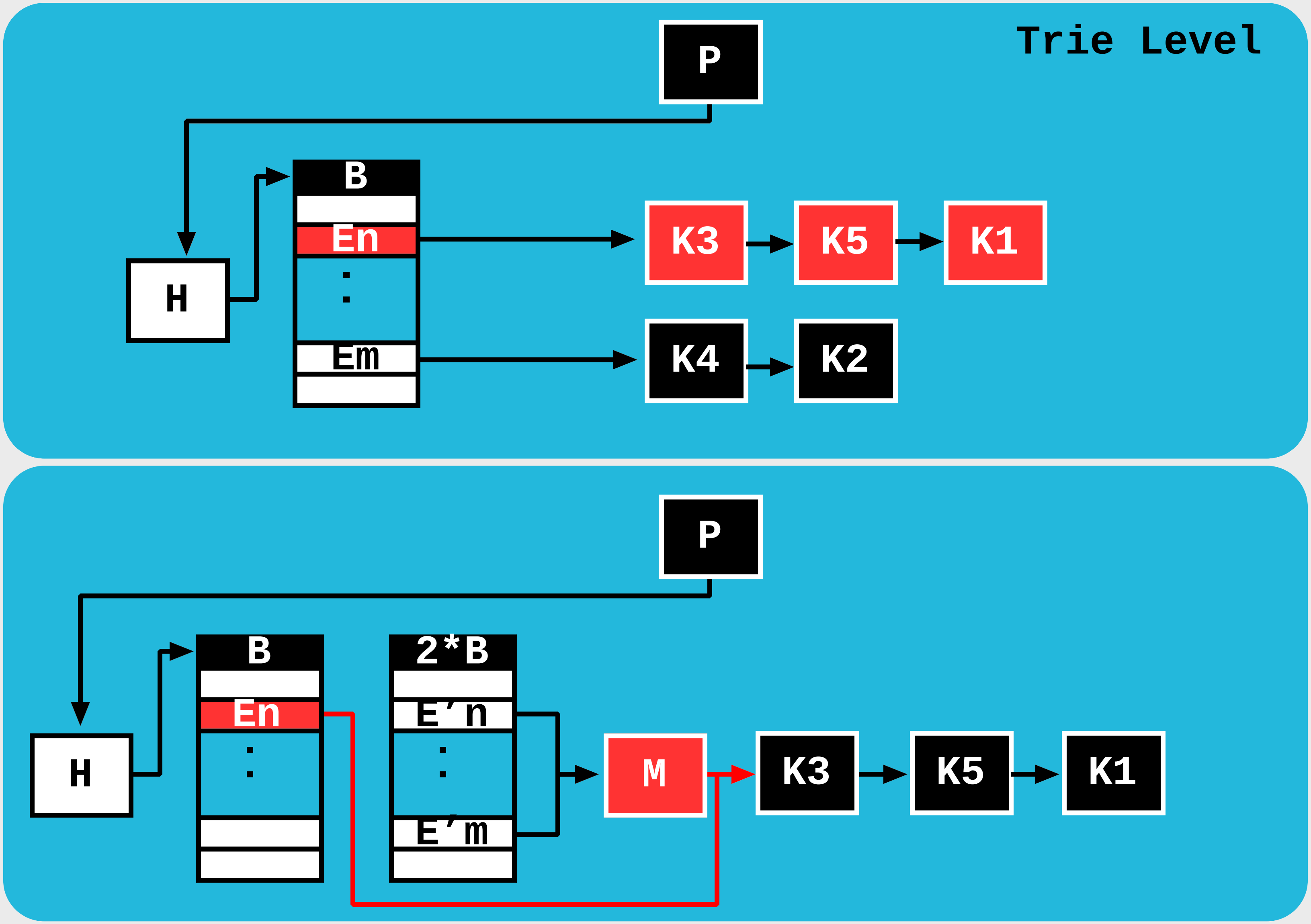
Lock-Free Tries - The First Expansion



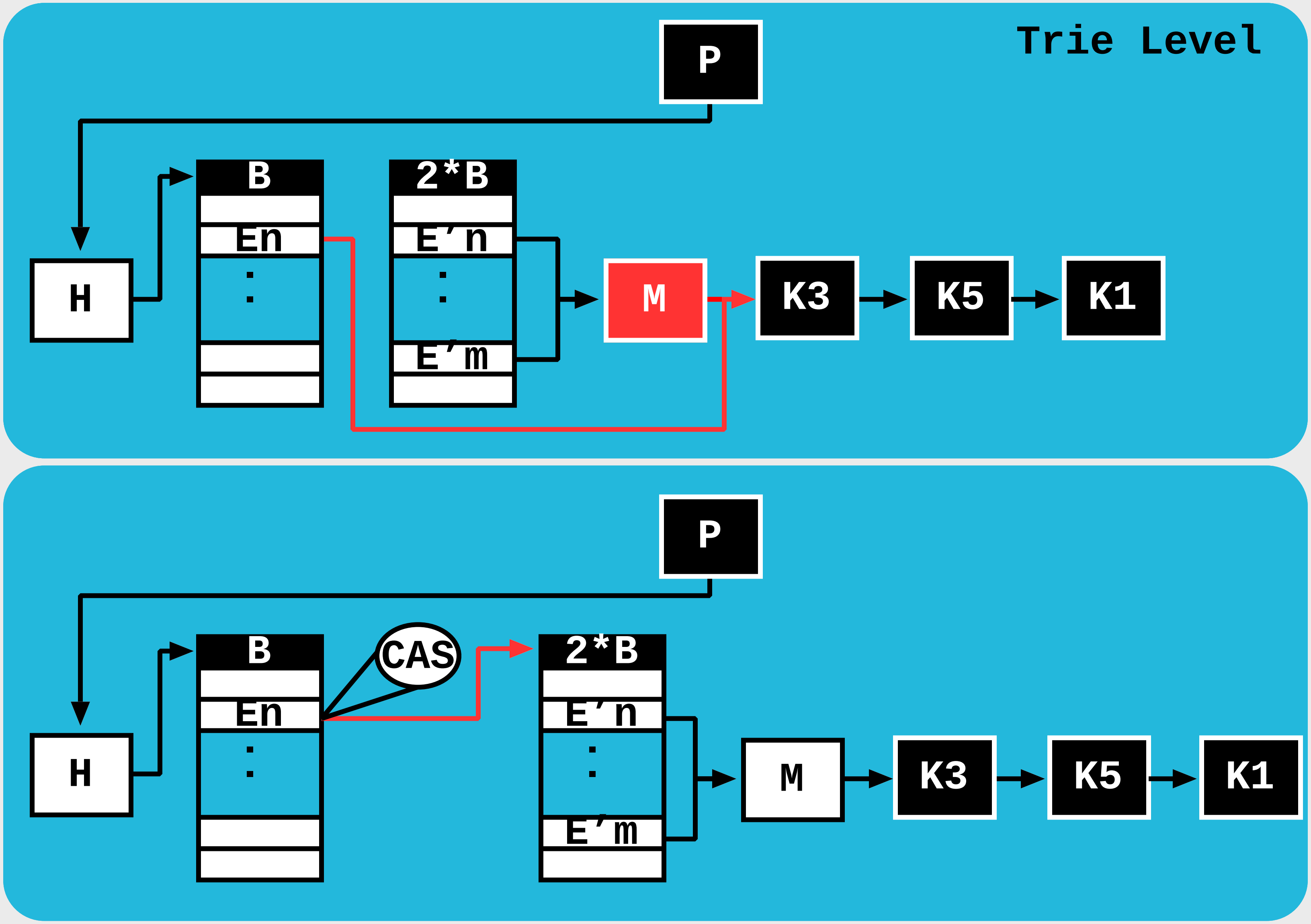
Lock-Free Tries - The Second Expansion



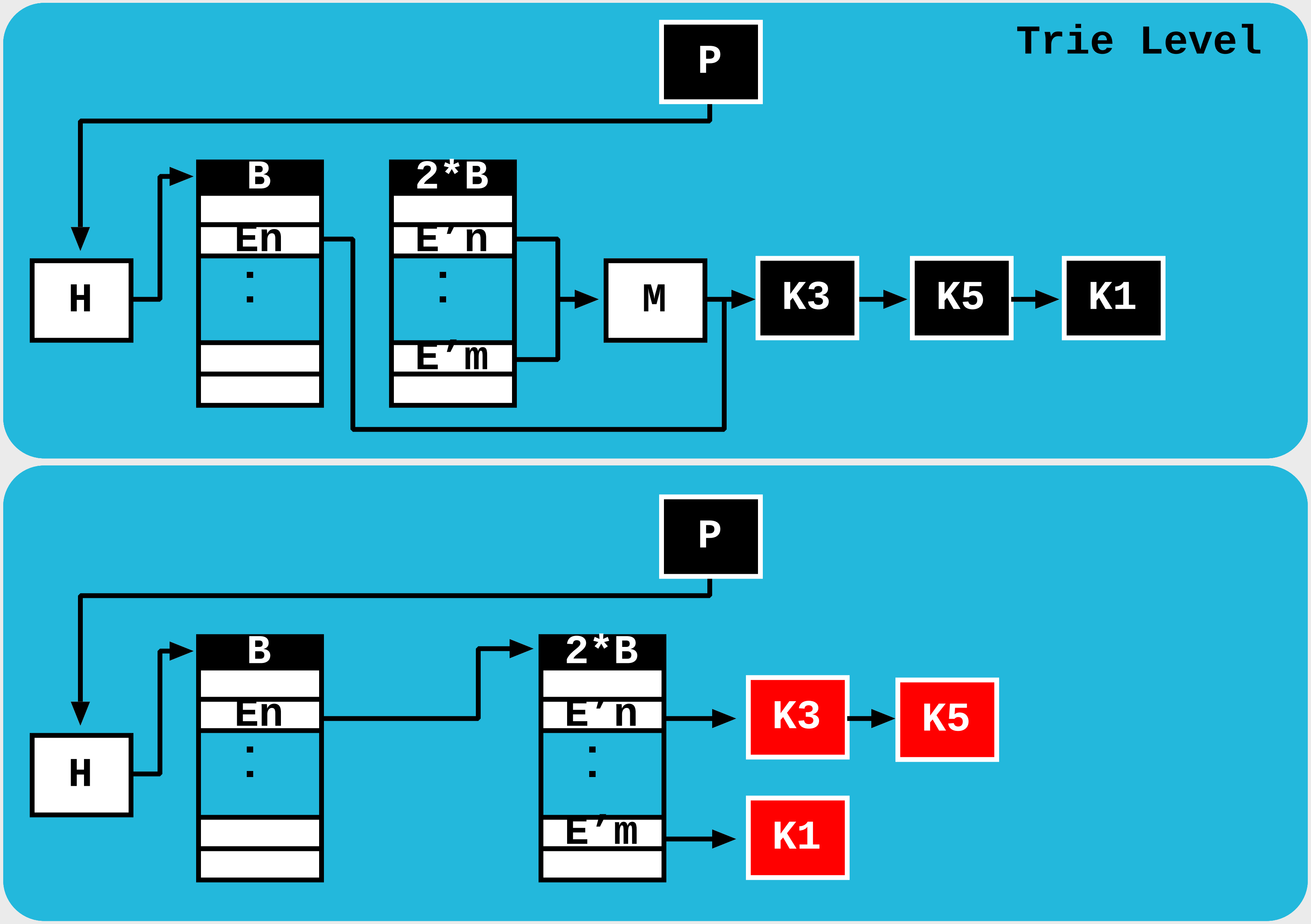
Lock-Free Tries - The Second Expansion



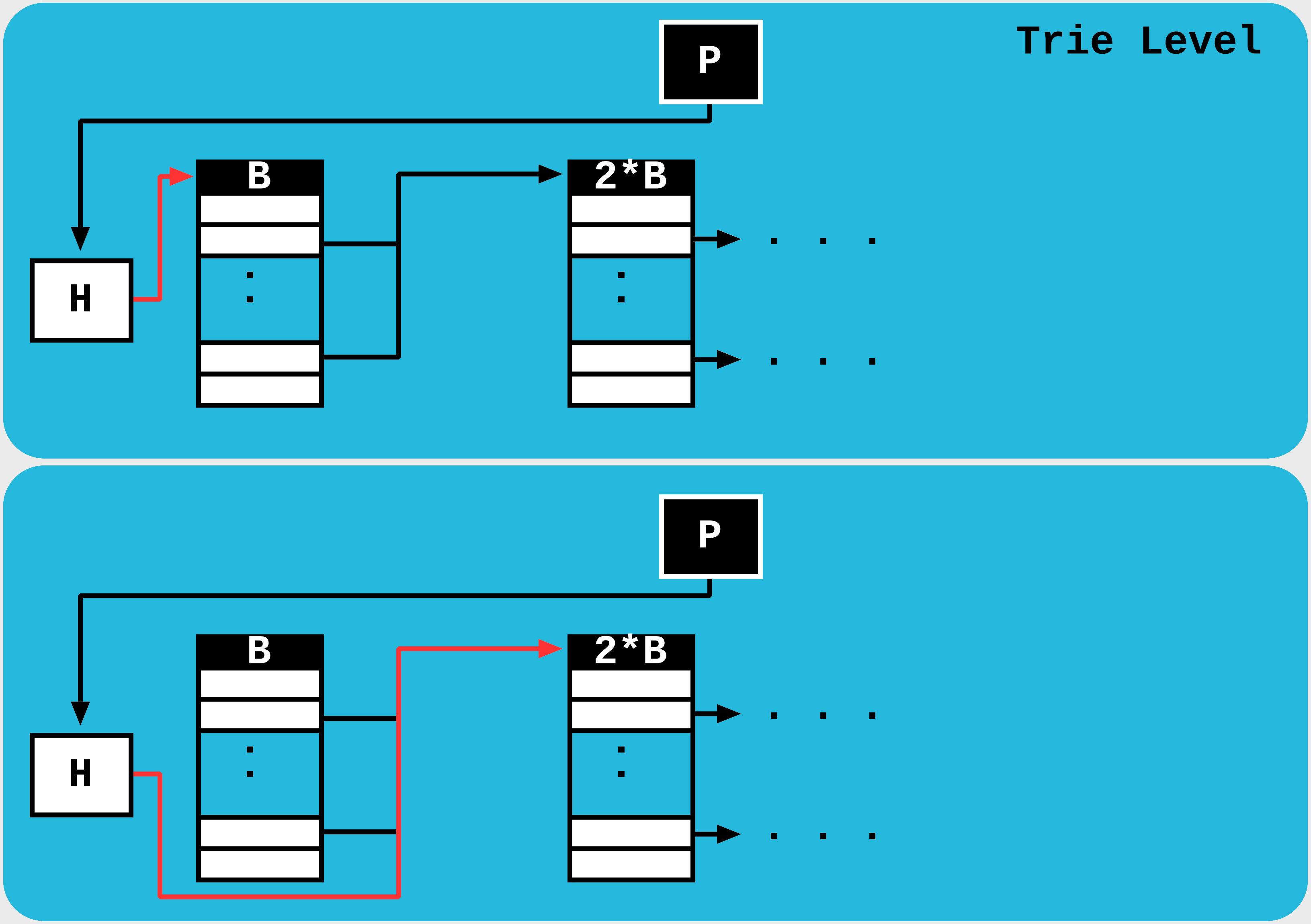
Lock-Free Tries - The Second Expansion



Lock-Free Tries - The Second Expansion



Lock-Free Tries - The Second Expansion



Lock-Free Tries - Resume

- **Avoids** the usage of **locks**:
  - **Reduces** the size of the nodes.
  - **Avoids problems** associated with **locks**:
    - Contention.
    - Convoying.
    - Priority inversion.

Lock-Free Tries - Resume

- **Avoids** the usage of **locks**:
  - **Reduces** the size of the nodes.
  - **Avoids problems** associated with **locks**:
    - Contention.
    - Convoying.
    - Priority inversion.

- The **create** and **expand** operations of the **hashing mechanism**:
  - **Does not lock** the **search** operation.
  - **Allow** the **concurrent insertion** of new nodes.

- **Different** nodes can be **inserted simultaneously** in **different bucket entries**.
  - Previous models **locked** all bucket entries to insert a new node.

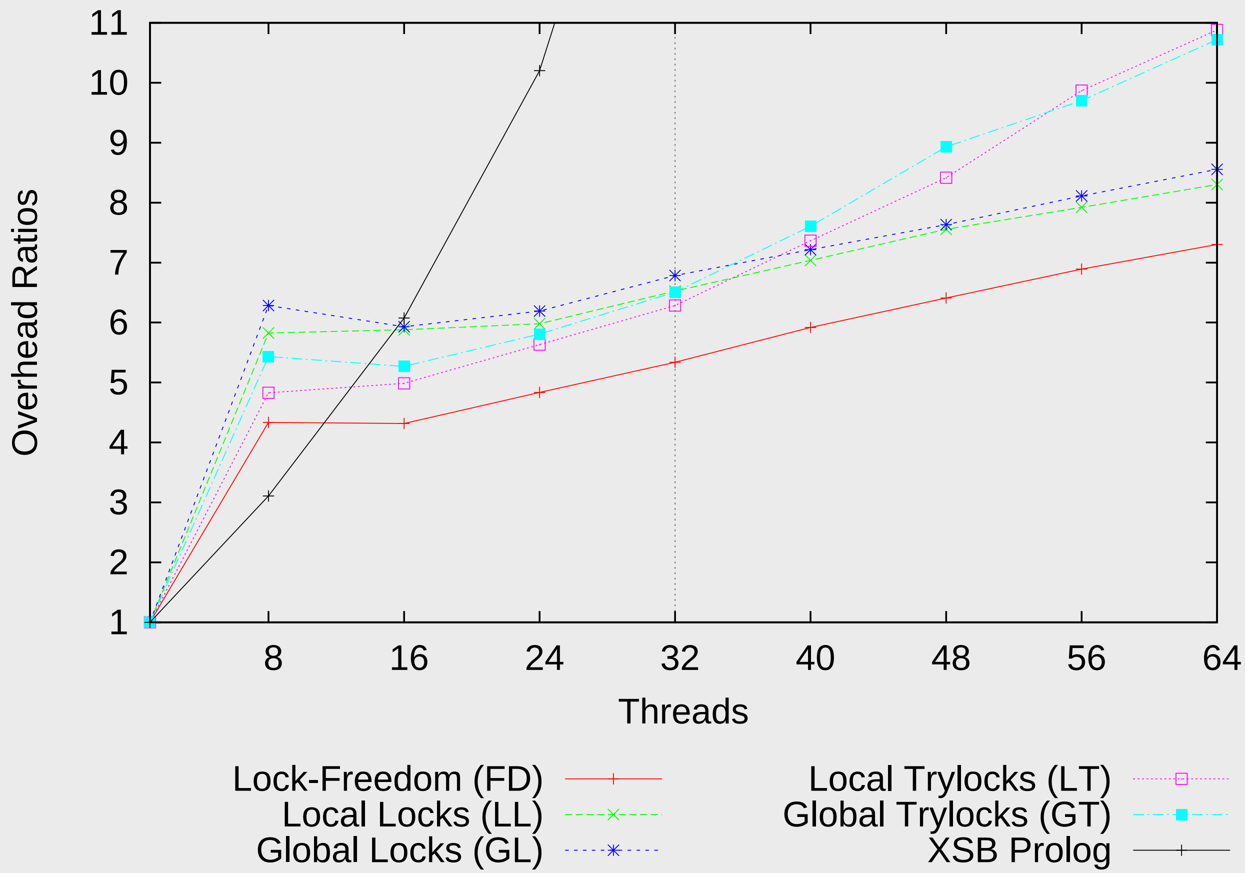
Experimental Results - Benchmark Statistics

Characteristics Of The Benchmarks: 1 Working Thread

Bench	Tabled Subgoals			Tabled Answers			
	Calls	Trie Nodes	Trie Depth	Unique	Repeated	Trie Nodes	Trie Depth
Model Checking							
IProto	1	6	5/5/5	134,361	385,423	1,554,896	4/51/67
Leader	1	5	4/4/4	1,728	574,786	41,788	15/80/97
Sieve	1	7	6/6/6	380	1,386,181	8,624	21/53/58
Large Joins							
Join2	1	6	5/5/5	2,476,099	0	2,613,660	5/5/5
Mondial	35	42	3/4/4	2,664	2,452,890	14,334	6/7/7
Path Left							
BTree	1	3	2/2/2	1,966,082	0	2,031,618	2/2/2
Pyramid	1	3	2/2/2	3,374,250	1,124,250	3,377,250	2/2/2
Cycle	1	3	2/2/2	4,000,000	2,000	4,002,001	2/2/2
Grid	1	3	2/2/2	1,500,625	4,335,135	1,501,851	2/2/2
Path Right							
BTree	131,071	262,143	2/2/2	3,801,094	0	3,997,700	1/2/2
Pyramid	3,000	6,001	2/2/2	6,745,501	2,247,001	6,751,500	1/2/2
Cycle	2,001	4,003	2/2/2	8,000,000	4,000	8,004,001	1/2/2
Grid	1,226	2,453	2/2/2	3,001,250	8,670,270	3,003,701	1/2/2

Experimental Results - Tabling Framework

- Comparison in a **32 Core AMD** machine. All threads execute the **same sub-computations** (**worst case** scenario). **Overhead ratios** comparing the **execution time** of multiple working threads against **the respective execution time** with one thread.



Lock-Free Tries - Summary

- We have presented a **first approach** for a **lock-free** trie data structures applied to the multithreaded tabled evaluation of logic programs:
  - Improve the **efficiency** of the concurrent search and insert operations.
  - The paper **On the Correctness and Efficiency of Lock-Free Expandable Tries for Tabled Logic Programs** discusses the most relevant **implementation details** and **proves the correctness** of the model.

Lock-Free Tries - Summary

- We have presented a **first approach** for a **lock-free** trie data structures applied to the multithreaded tabled evaluation of logic programs:
  - Improve the **efficiency** of the concurrent search and insert operations.
  - The paper **On the Correctness and Efficiency of Lock-Free Expandable Tries for Tabled Logic Programs** discusses the most relevant **implementation details** and **proves the correctness** of the model.

- Experimental results show that our approach can **effectively** reduce the **execution time** and **scale better**, when increasing the number of threads, than the original lock-based designs.

Lock-Free Tries - Summary

- We have presented a **first approach** for a **lock-free** trie data structures applied to the multithreaded tabled evaluation of logic programs:
  - Improve the **efficiency** of the concurrent search and insert operations.
  - The paper **On the Correctness and Efficiency of Lock-Free Expandable Tries for Tabled Logic Programs** discusses the most relevant **implementation details** and **proves the correctness** of the model.
- Experimental results show that our approach can **effectively** reduce the **execution time** and **scale better**, when increasing the number of threads, than the original lock-based designs.
- But... with **a deeper study** of this model we found that it **still has** some **concurrency problems**:
  - **Low dispersion** of the synchronization points
  - **False sharing** (memory cache secondary effects).

Lock-Free Hash Tries - Introduction

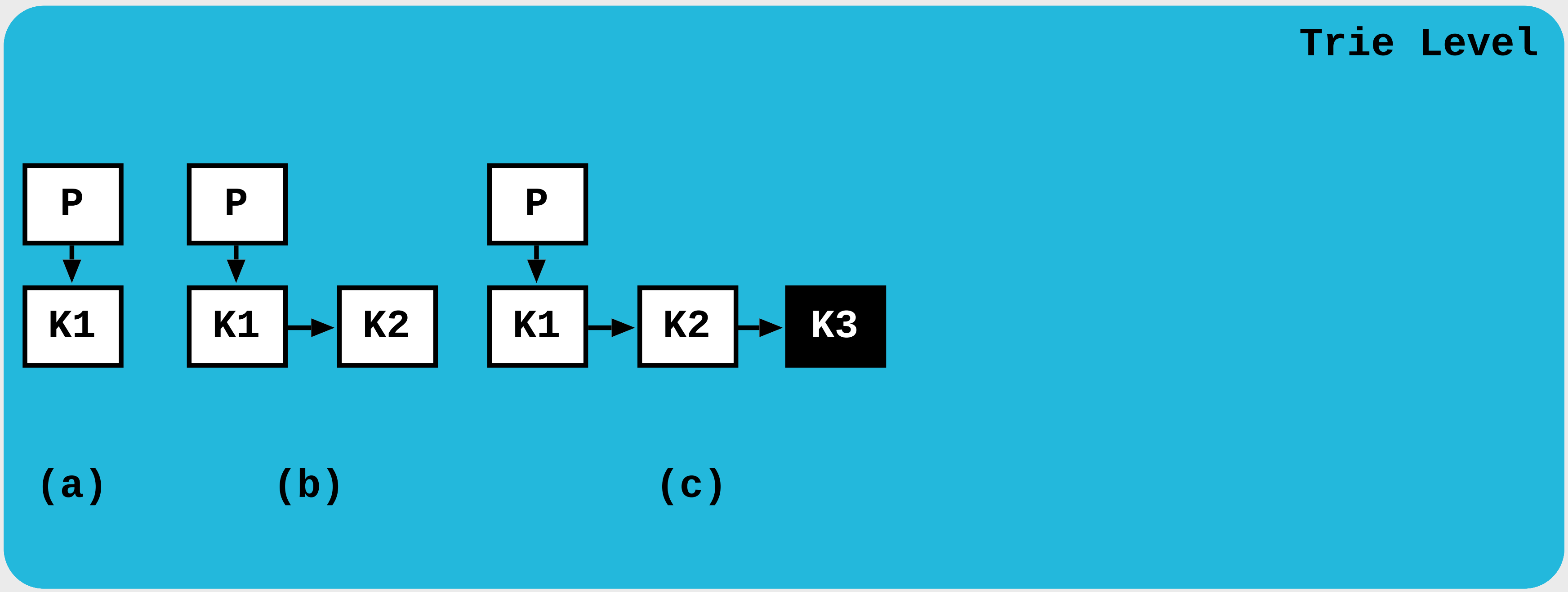
- A **trie level** is defined by a **parent (P)** node and at least one **child (K)** node.
- Only **lookup** and **insert** operations are executed.
- **Insertion** of new nodes is done in a **chain**, until a **threshold** is achieved and afterwards a **hashing system** is included in the trie level.

Lock-Free Hash Tries - Introduction

- A **trie level** is defined by a **parent (P)** node and at least one **child (K)** node.
- Only **lookup** and **insert** operations are executed.
- **Insertion** of new nodes is done in a **chain**, until a **threshold** is achieved and afterwards a **hashing system** is included in the trie level.

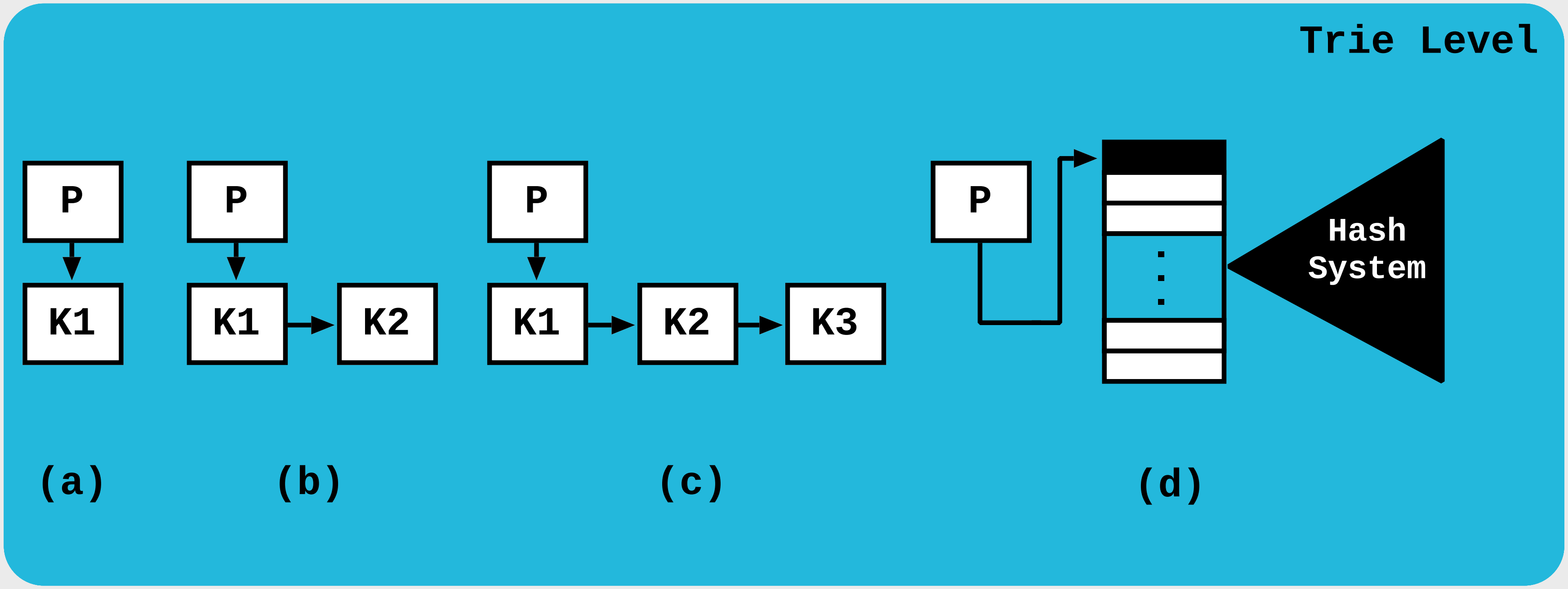
Lock-Free Hash Tries - Introduction

- A **trie level** is defined by a **parent (P)** node and at least one **child (K)** node.
- Only **lookup** and **insert** operations are executed.
- **Insertion** of new nodes is done in a **chain**, until a **threshold** is achieved and afterwards a **hashing system** is included in the trie level.



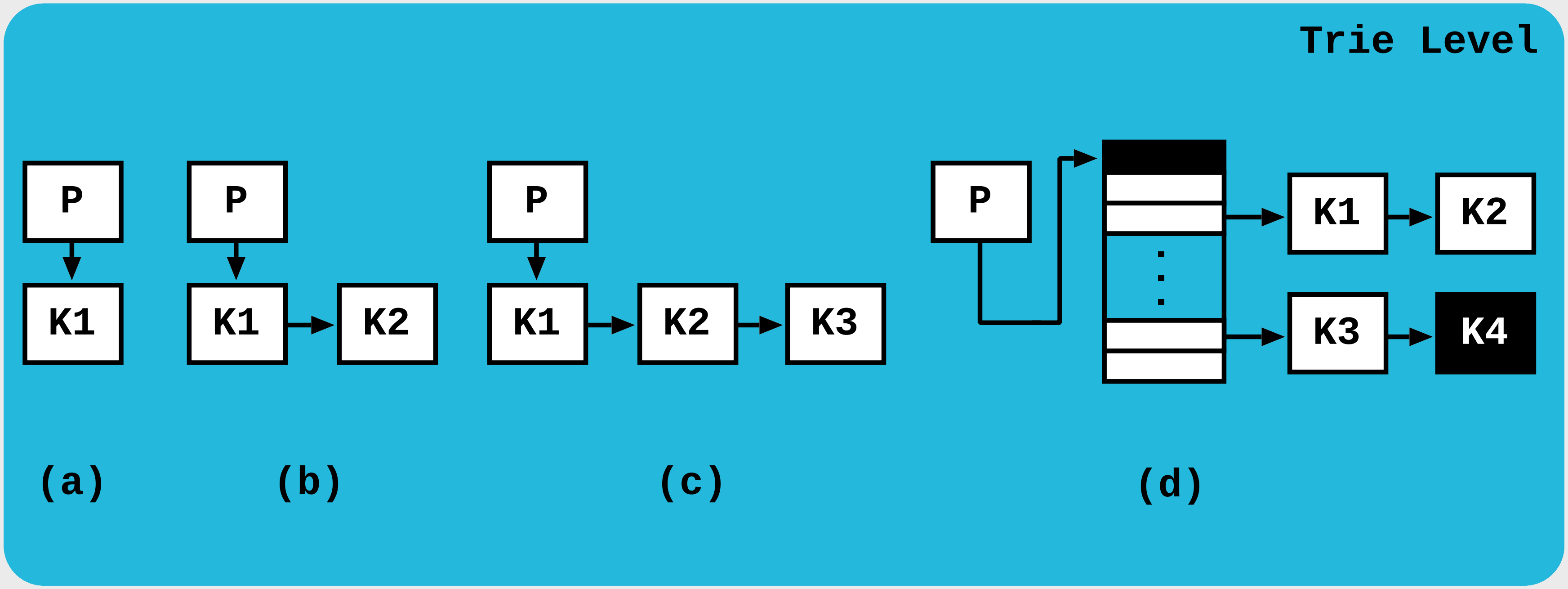
Lock-Free Hash Tries - Introduction

- A **trie level** is defined by a **parent (P)** node and at least one **child (K)** node.
- Only **lookup** and **insert** operations are executed.
- **Insertion** of new nodes is done in a **chain**, until a **threshold** is achieved and afterwards a **hashing system** is included in the trie level.



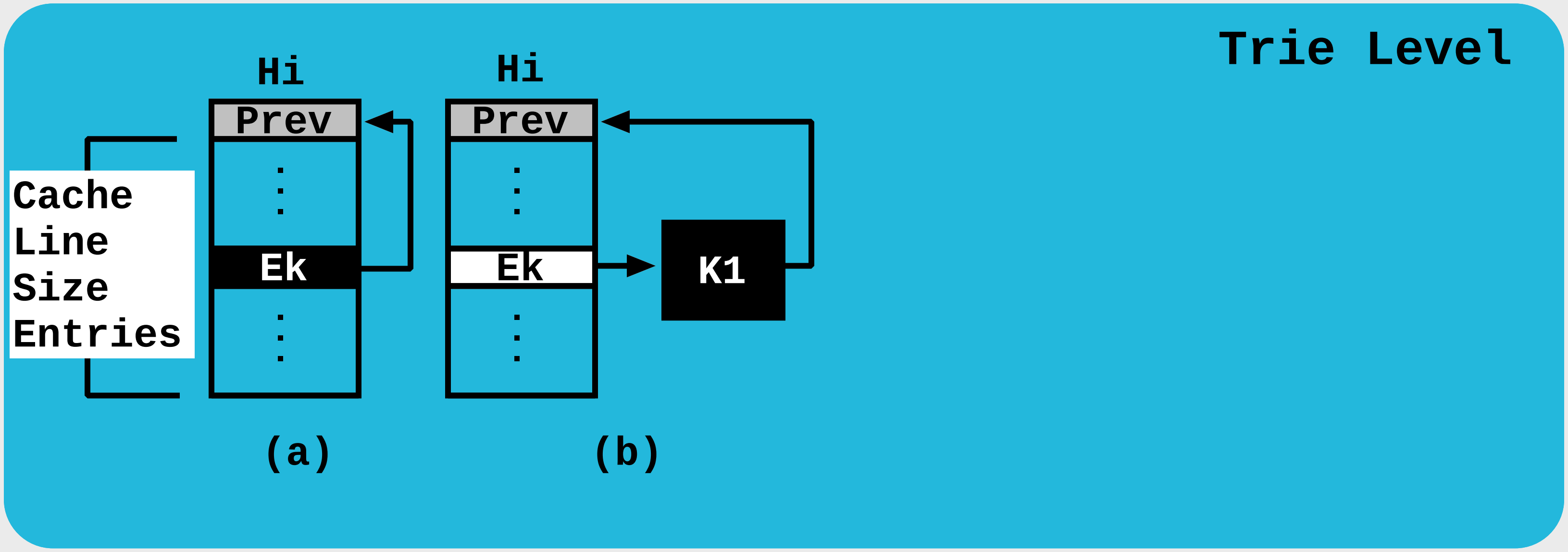
Lock-Free Hash Tries - Introduction

- A **trie level** is defined by a **parent (P)** node and at least one **child (K)** node.
- Only **lookup** and **insert** operations are executed.
- **Insertion** of new nodes is done in a **chain**, until a **threshold** is achieved and afterwards a **hashing system** is included in the trie level.

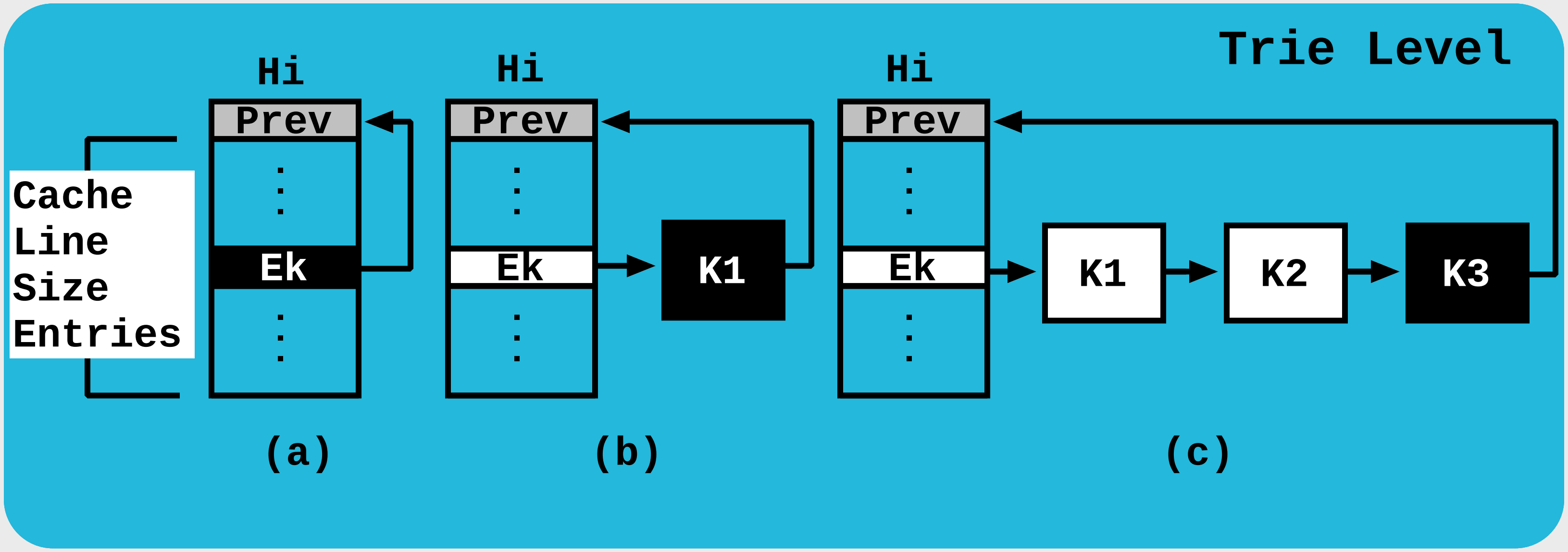


Lock-Free Hash Tries - Key Ideas

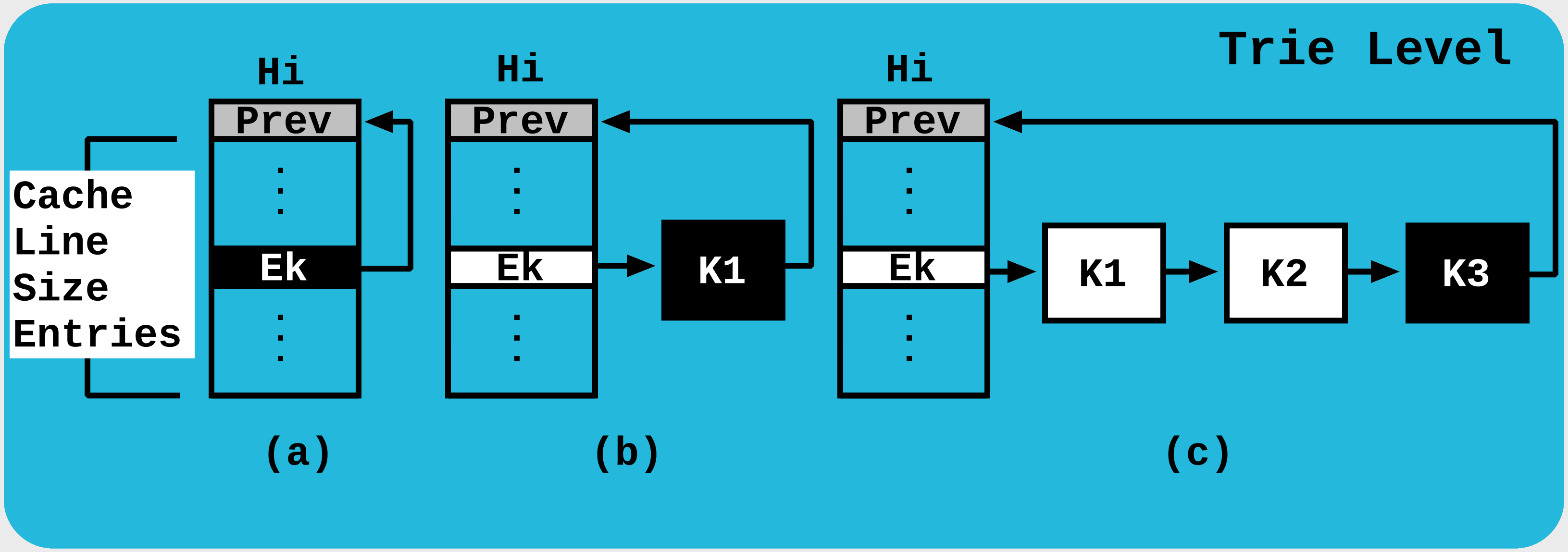
Lock-Free Hash Tries - Key Ideas

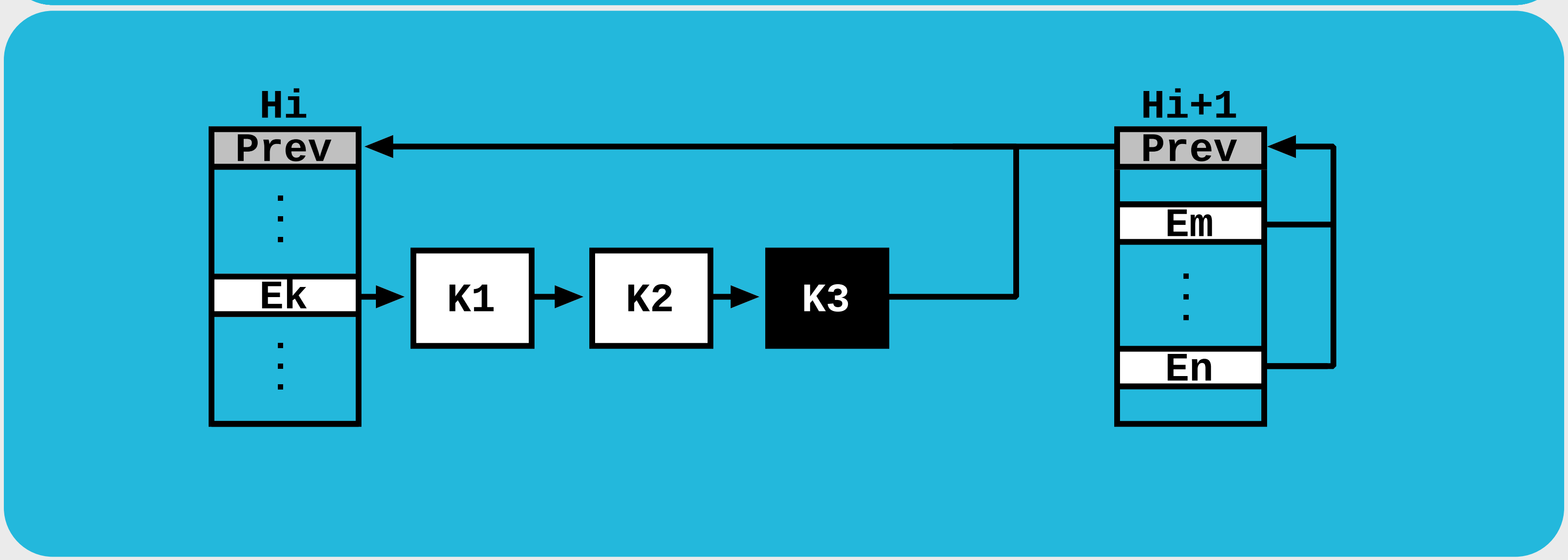


Lock-Free Hash Tries - Key Ideas

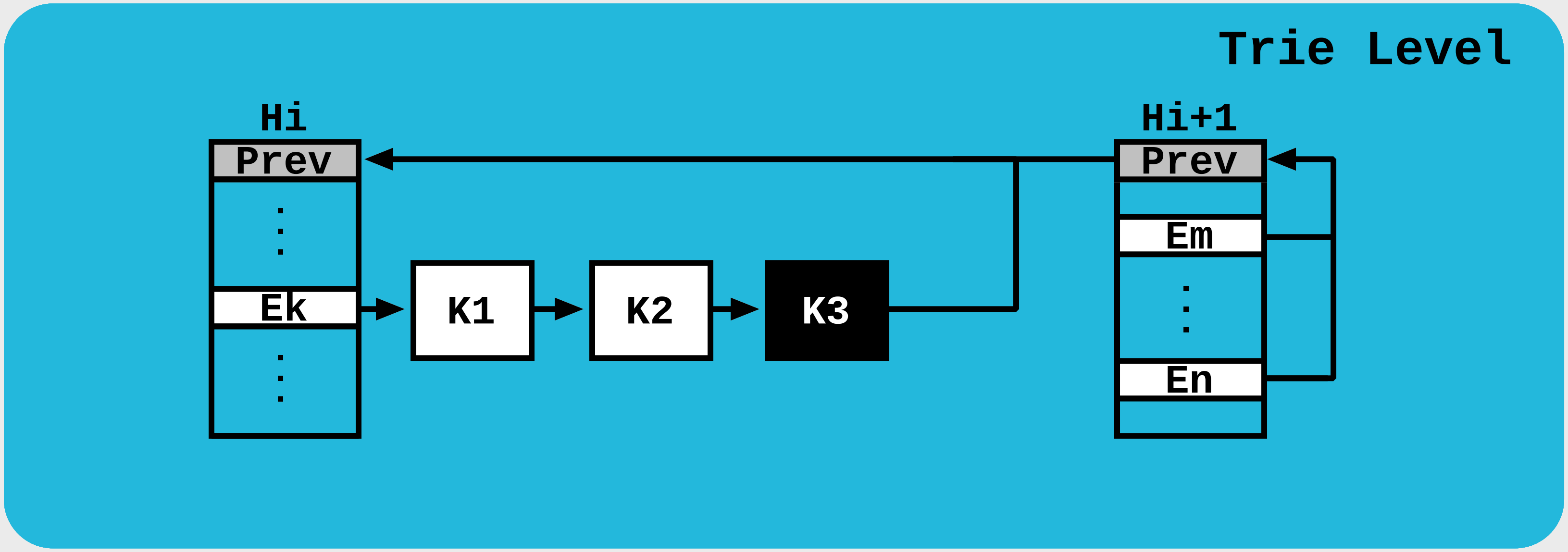


Lock-Free Hash Tries - Key Ideas

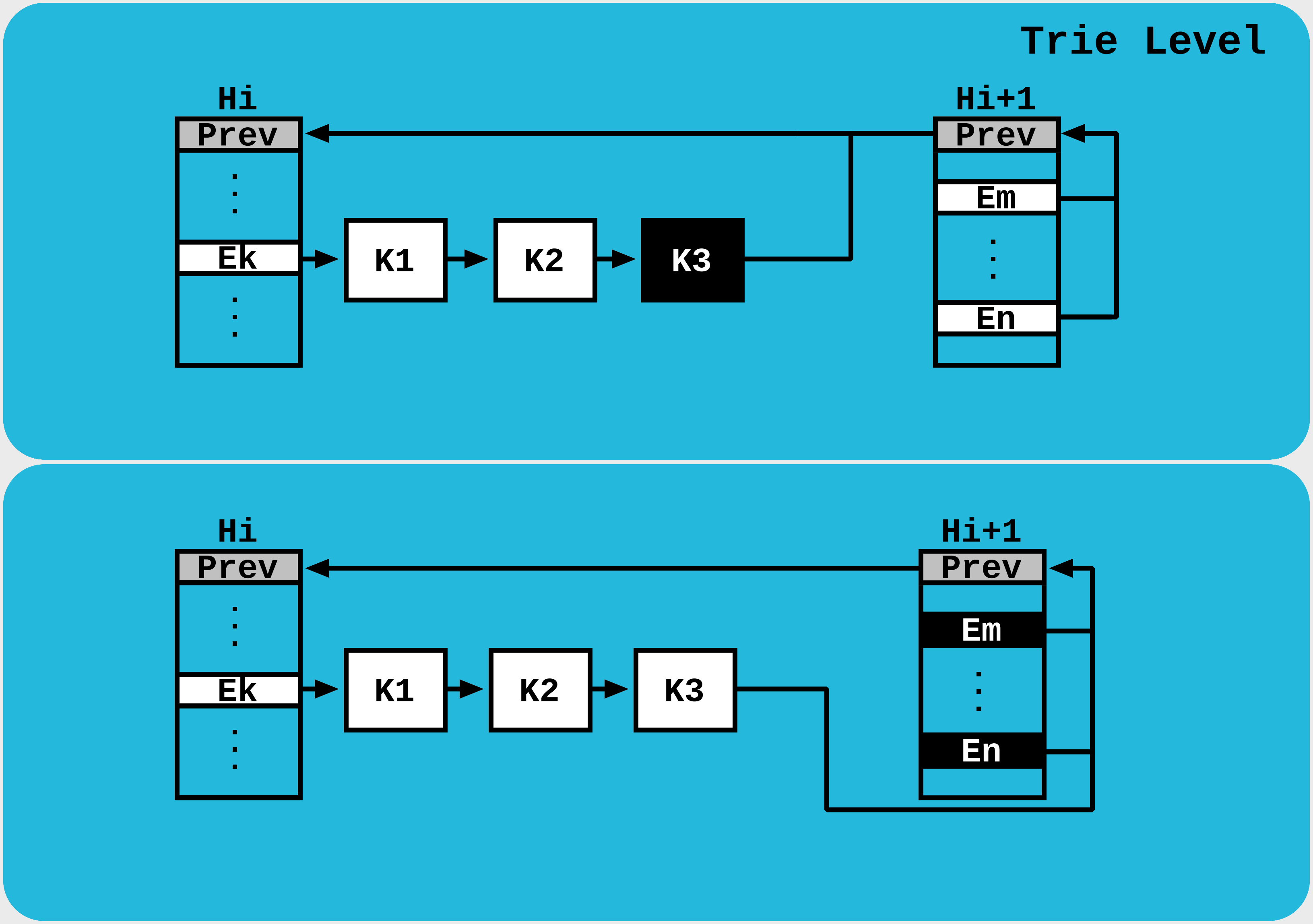




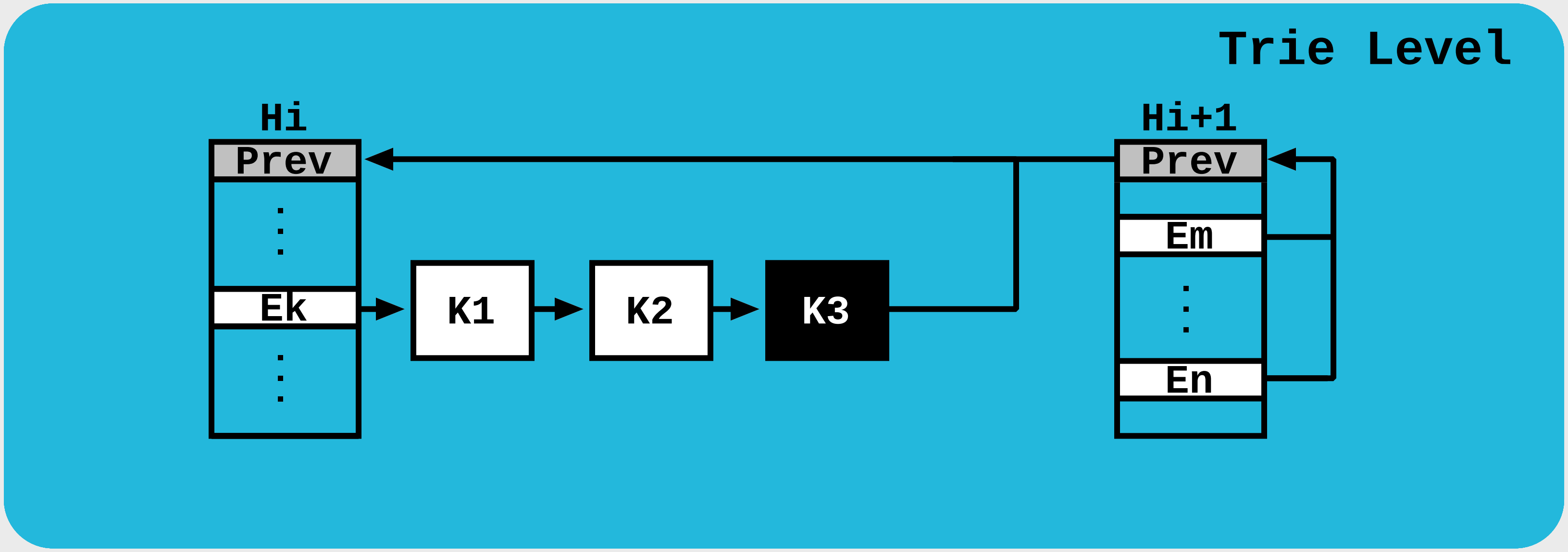
Lock-Free Hash Tries - Key Ideas

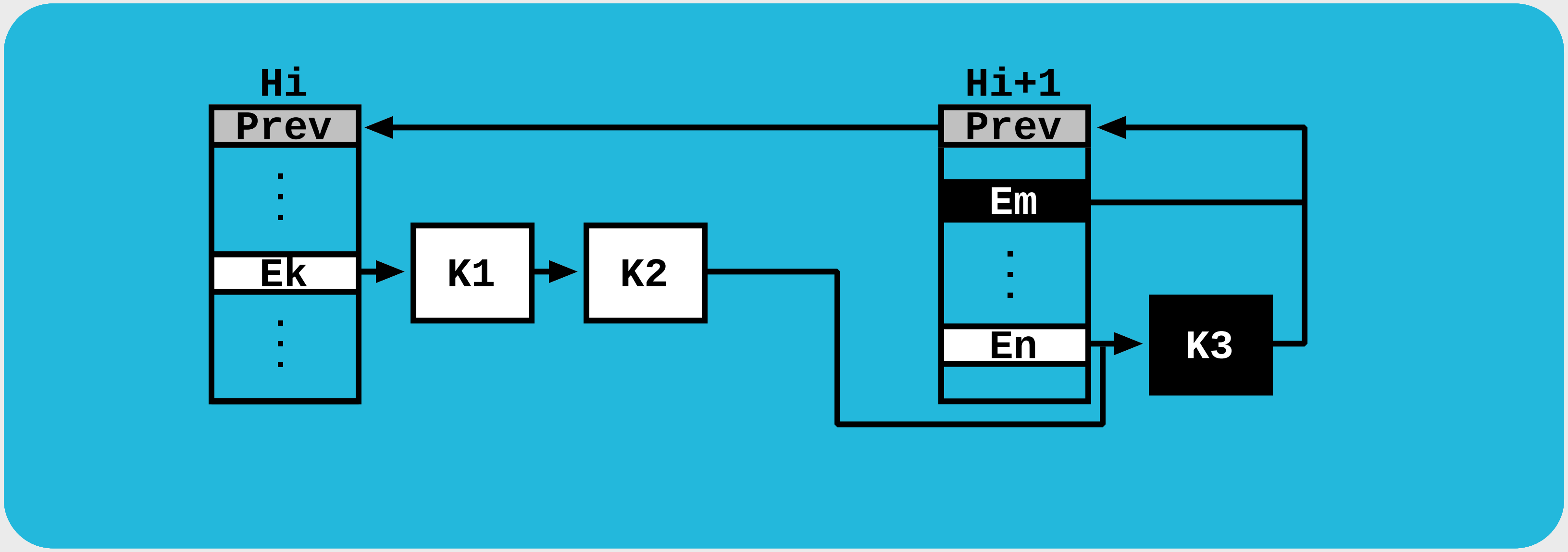


Lock-Free Hash Tries - Key Ideas

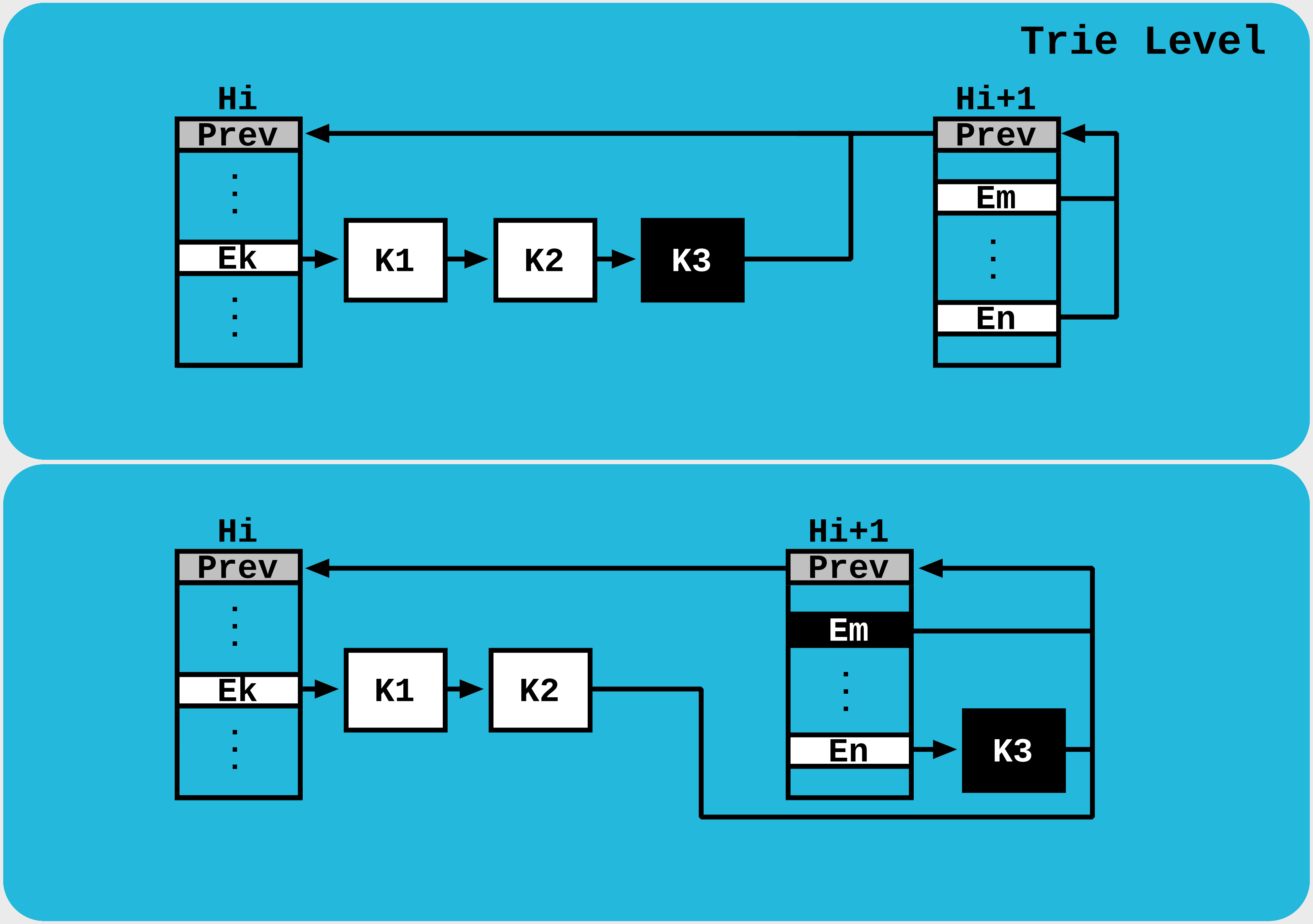


Lock-Free Hash Tries - Key Ideas

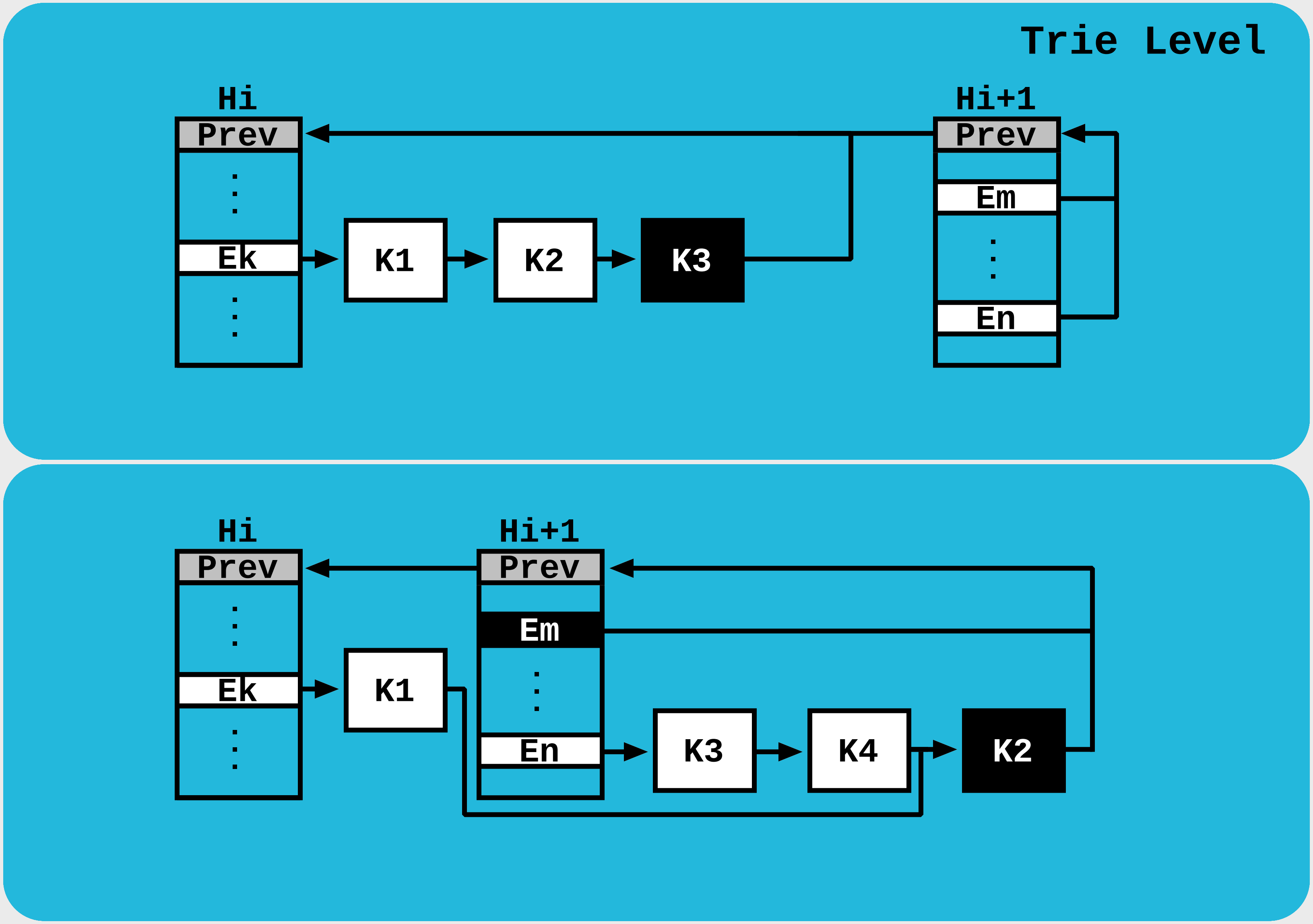




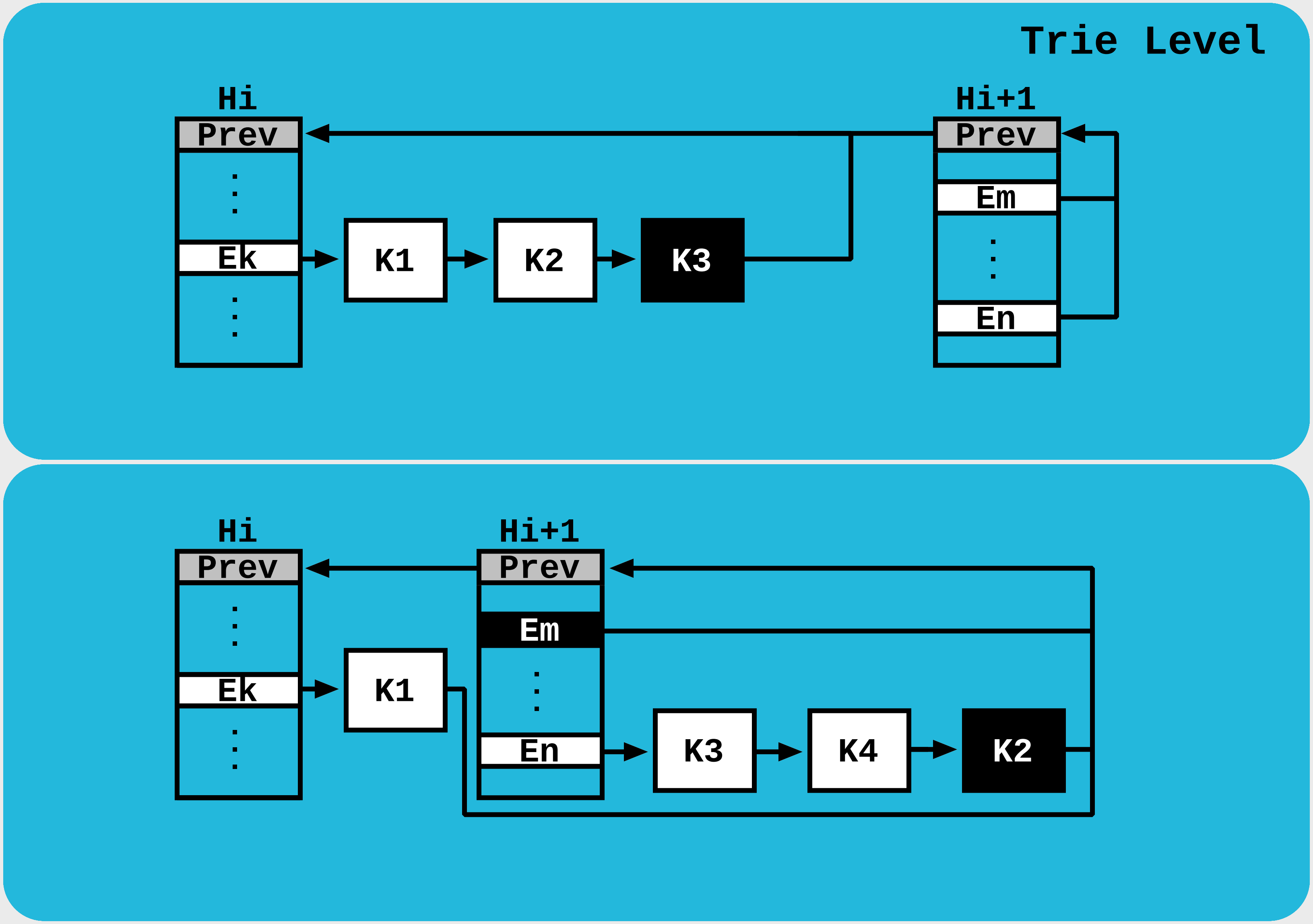
Lock-Free Hash Tries - Key Ideas



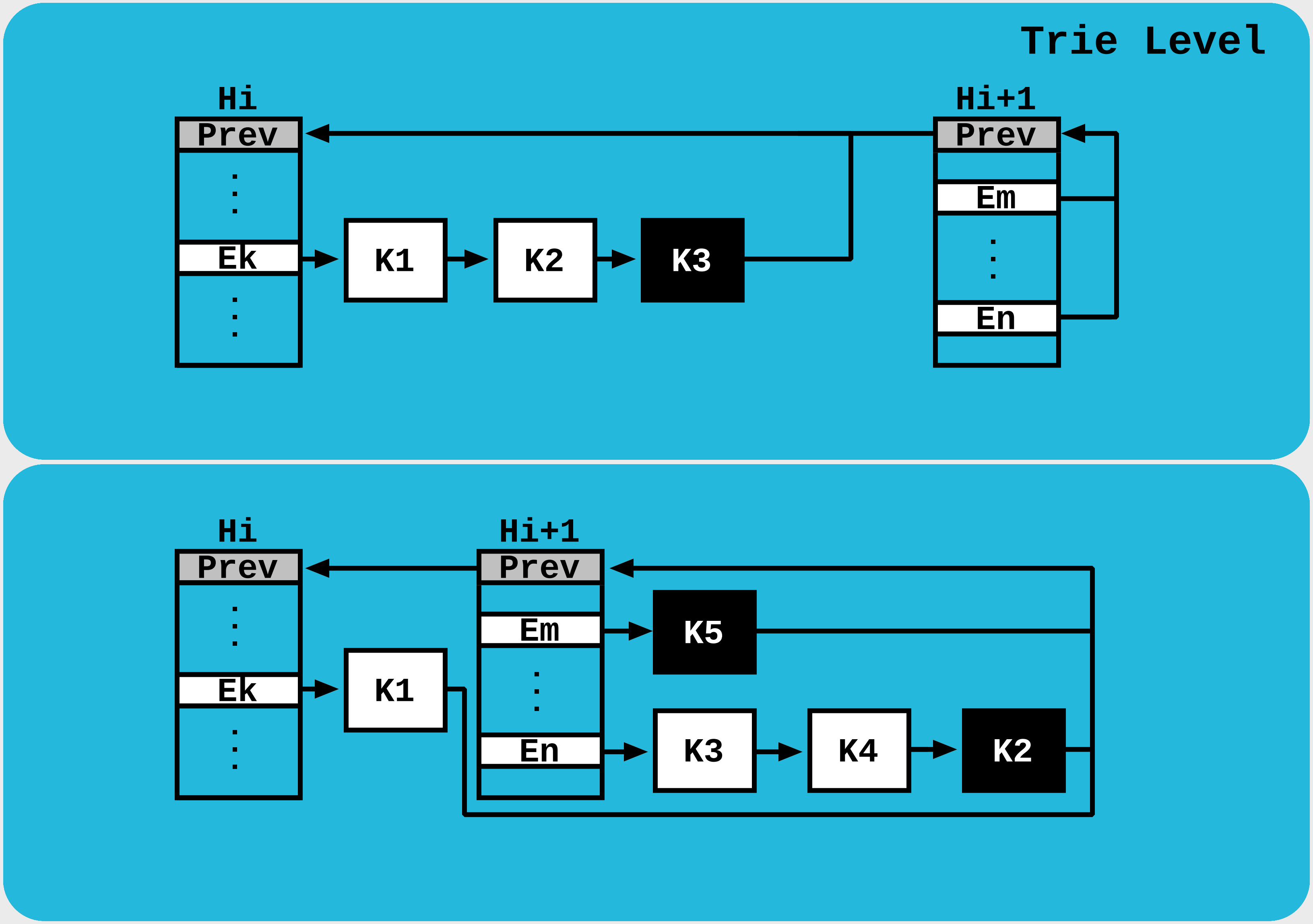
Lock-Free Hash Tries - Key Ideas



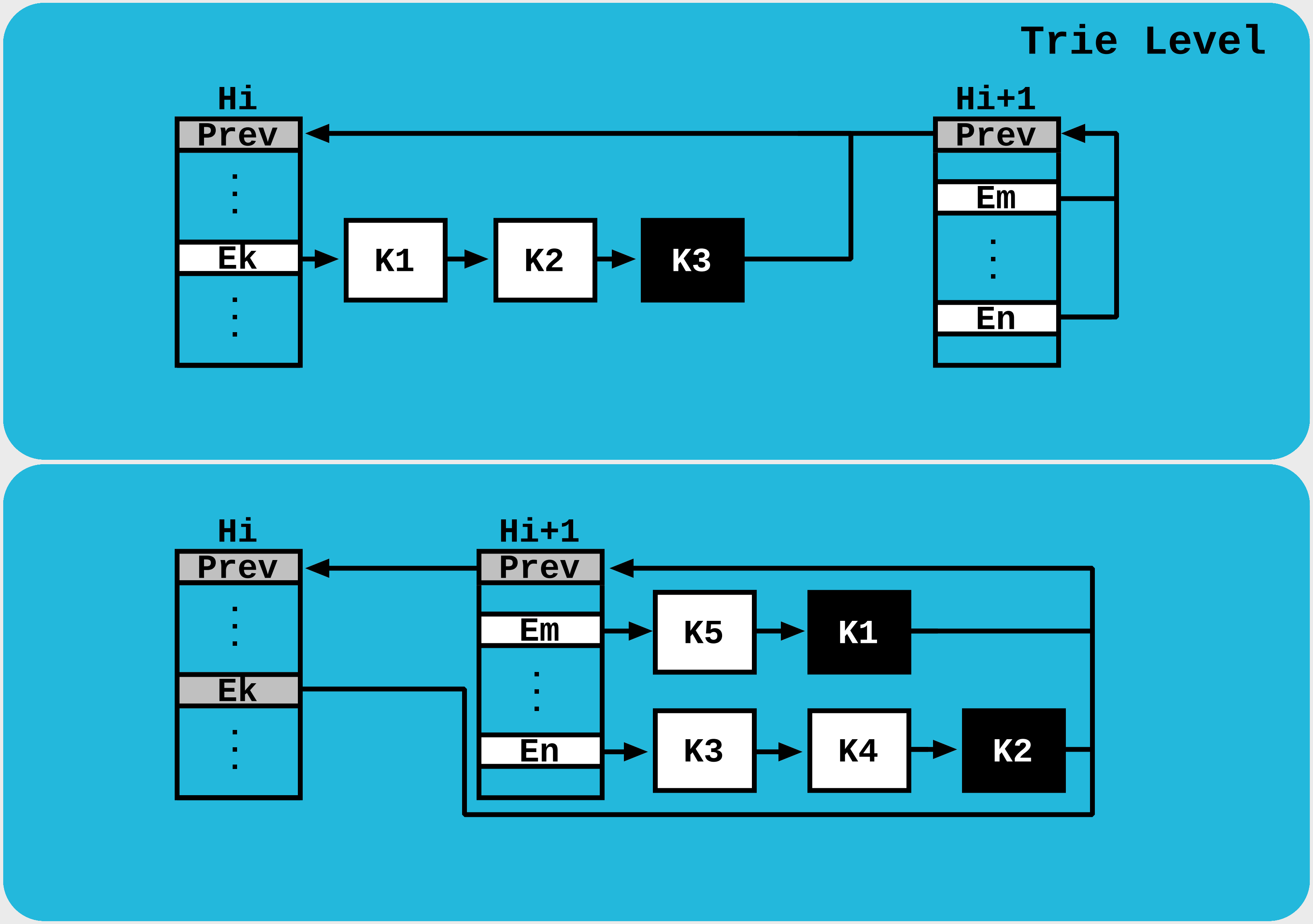
Lock-Free Hash Tries - Key Ideas



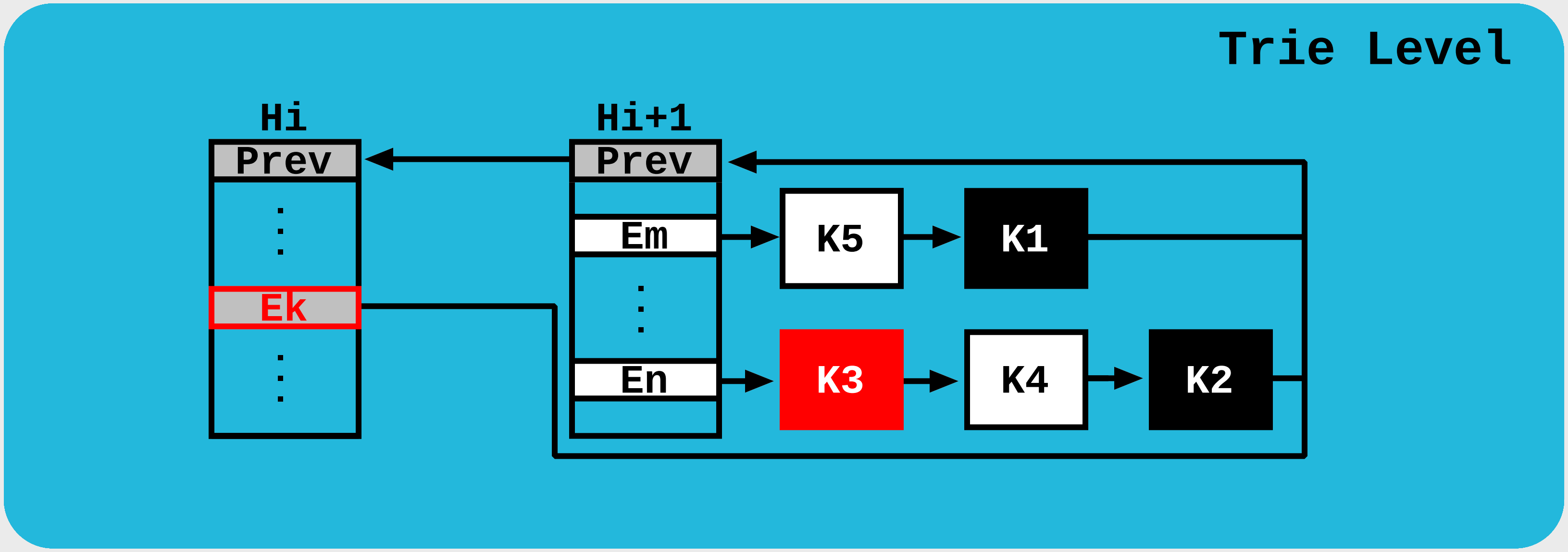
Lock-Free Hash Tries - Key Ideas

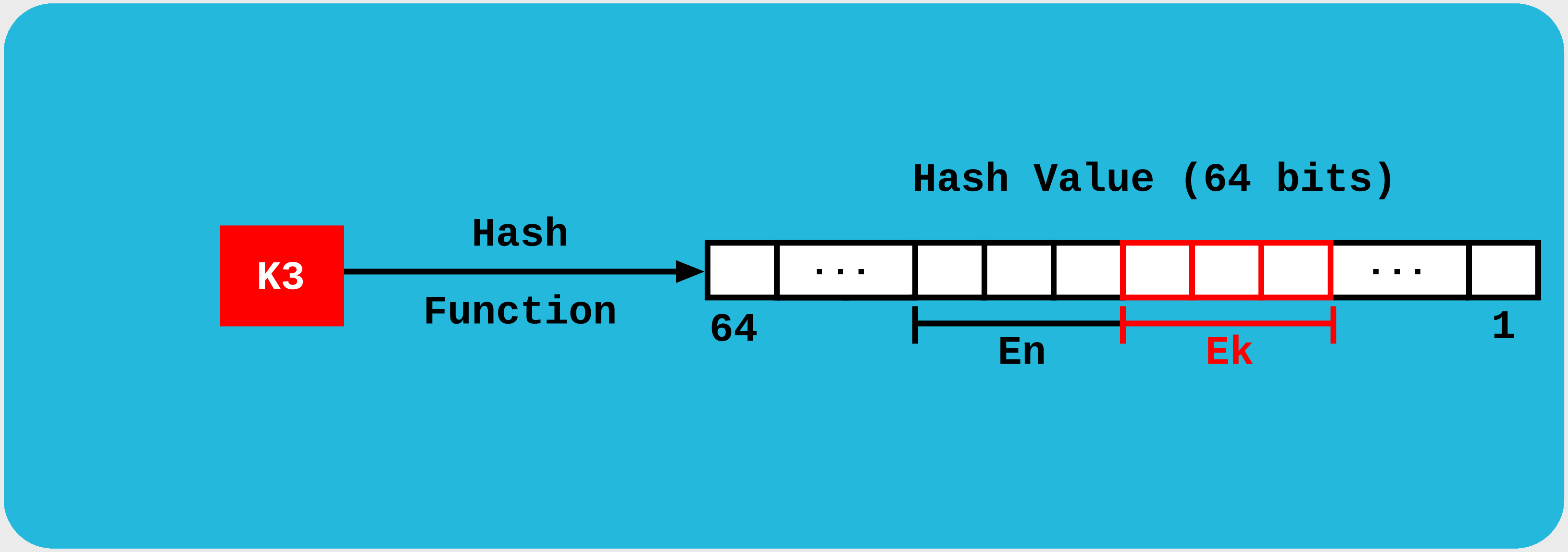


Lock-Free Hash Tries - Key Ideas

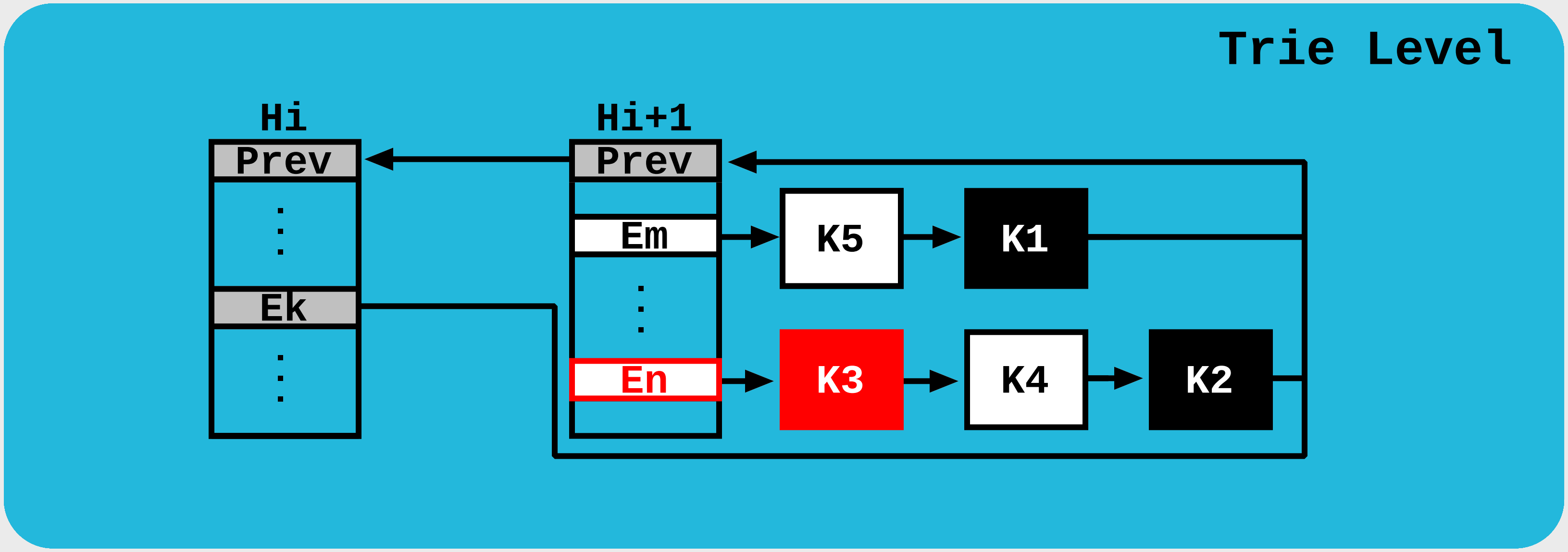


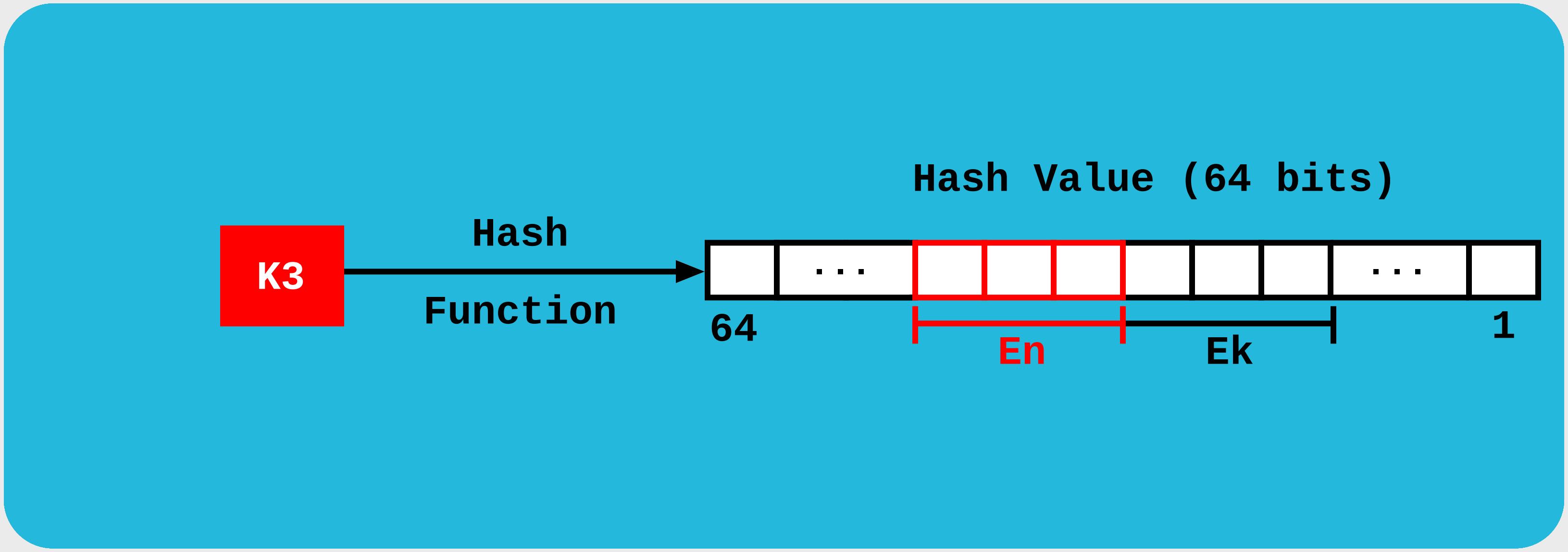
Lock-Free Hash Tries - Key Ideas





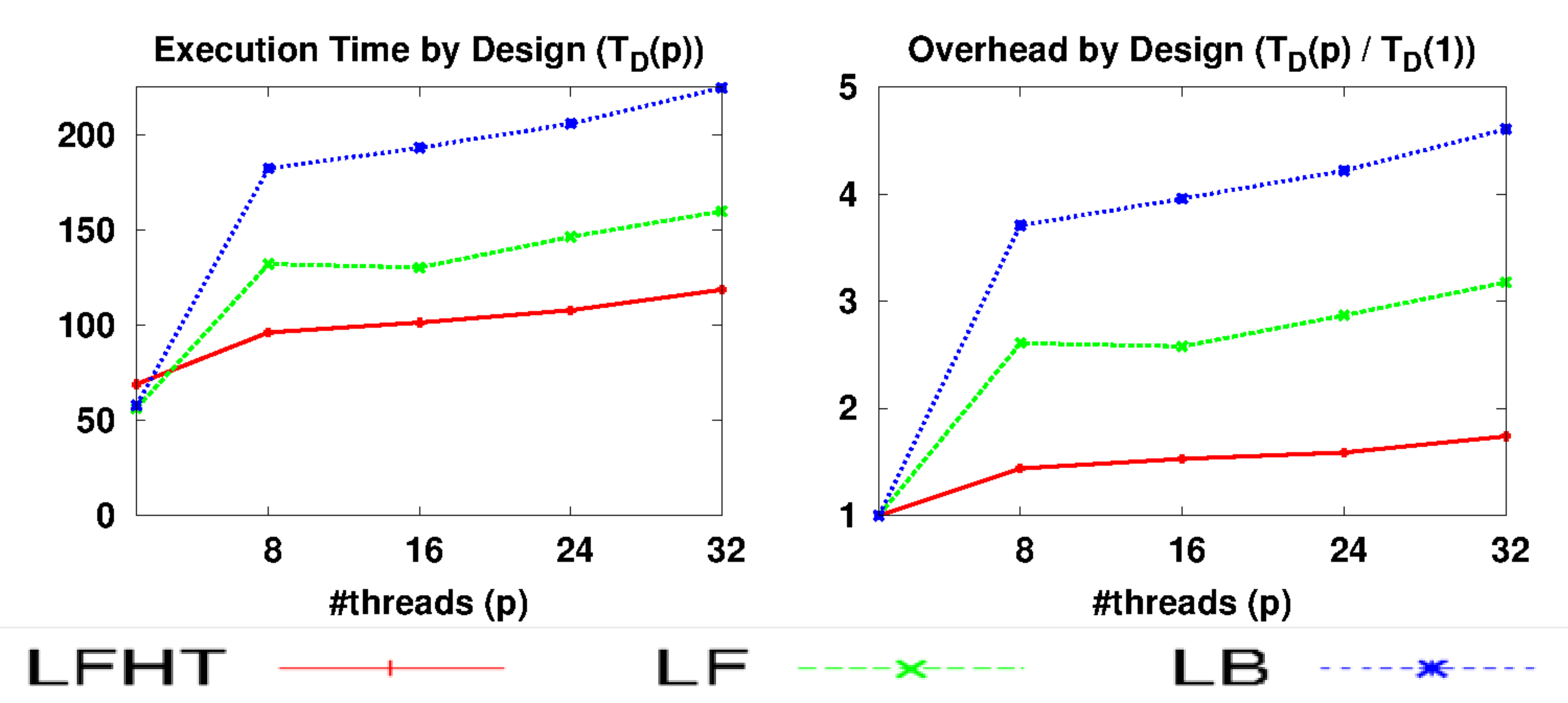
Lock-Free Hash Tries - Key Ideas





Experimental Results - Tabling Framework

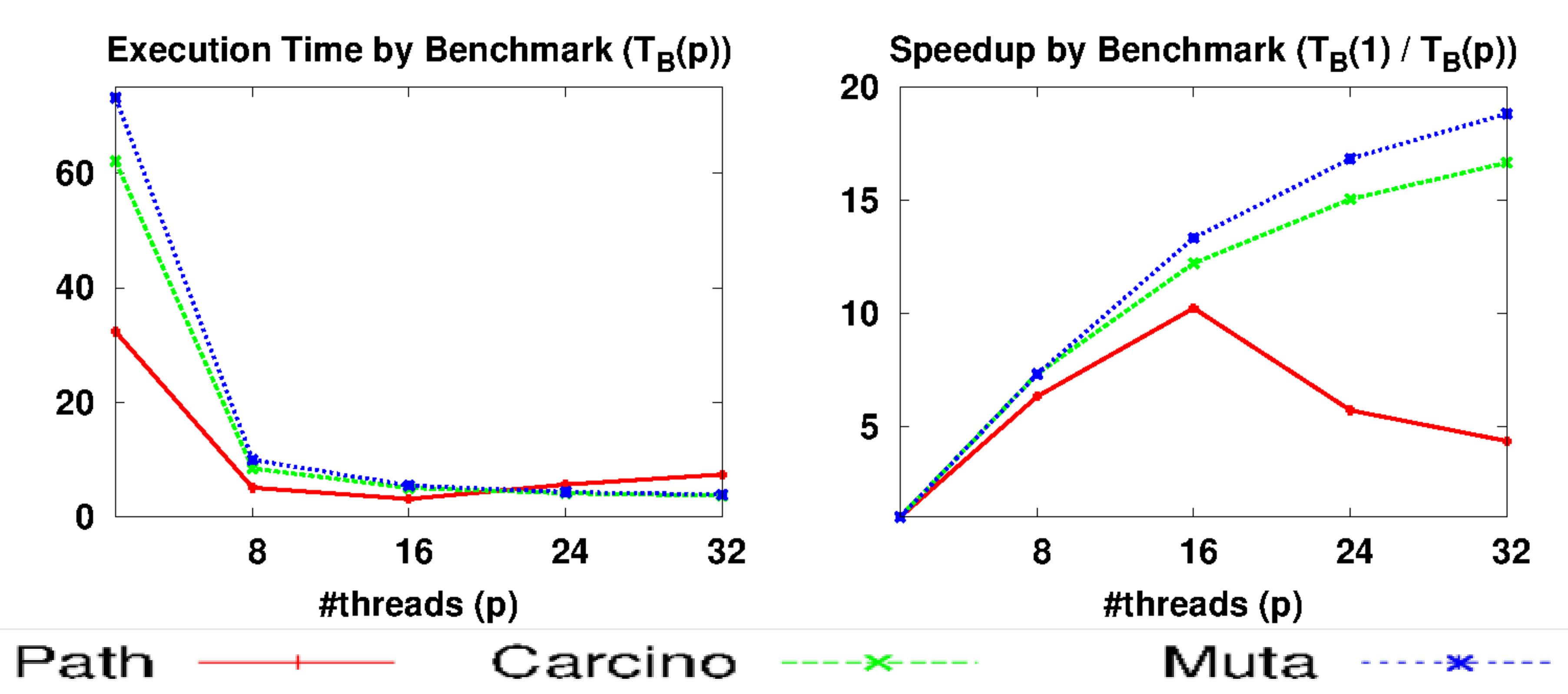
- Comparison in a **32 Core AMD** machine. All threads execute the **same sub-computations** (**worst case** scenario).



- **LFHT Lock-Free Hash Tries** - **LF Lock-Free (old approach)**
- **LB Lock-Based (old approach)**

Experimental Results - Tabling Framework

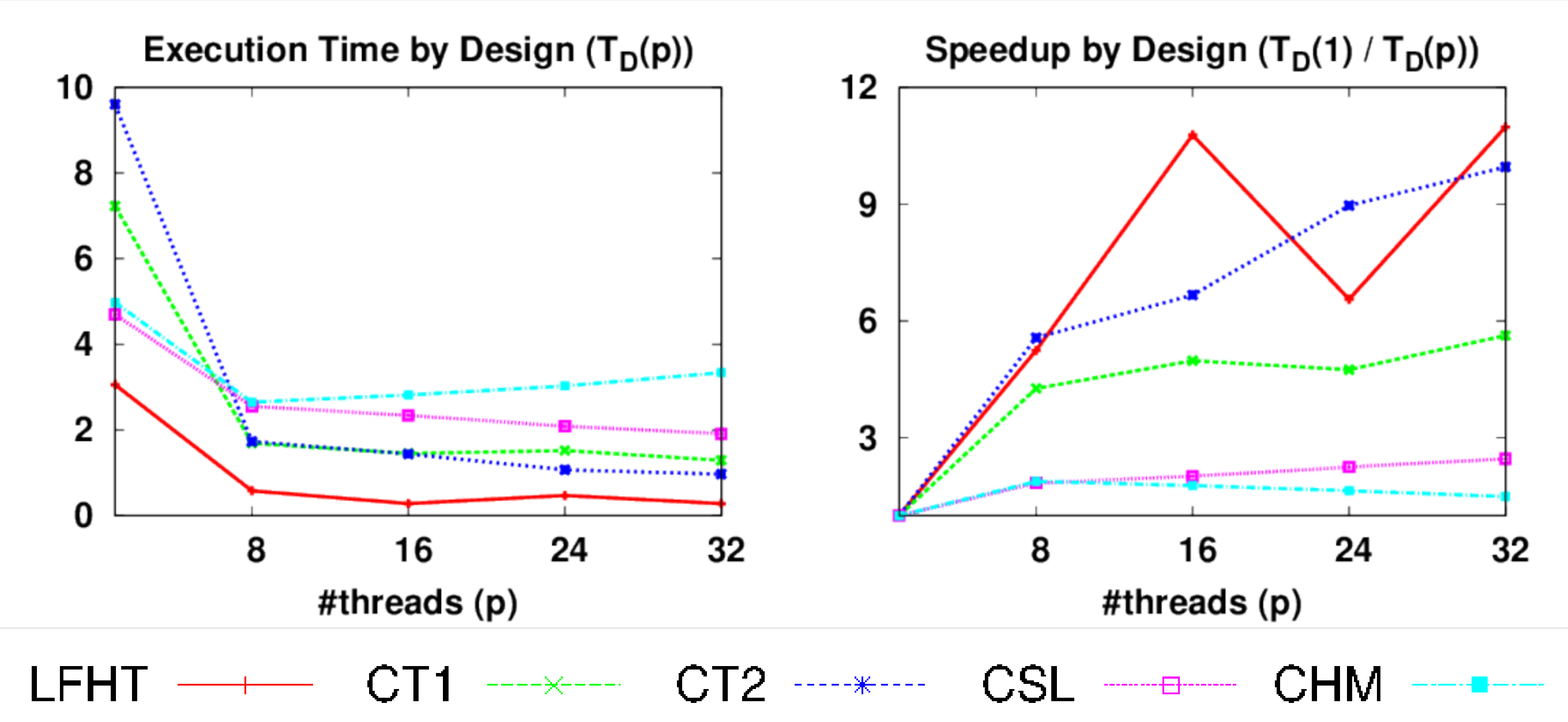
- Comparison in a **32 Core AMD** machine. All threads execute **different sub-computations** (**LFHT Lock-Free Hash Tries**).



- **Path Path problem using a graph with a grid configuration**
- **Carcino / Muta (genesis) Inductive Logic Programing Benchmarks**

Experimental Results - External Framework

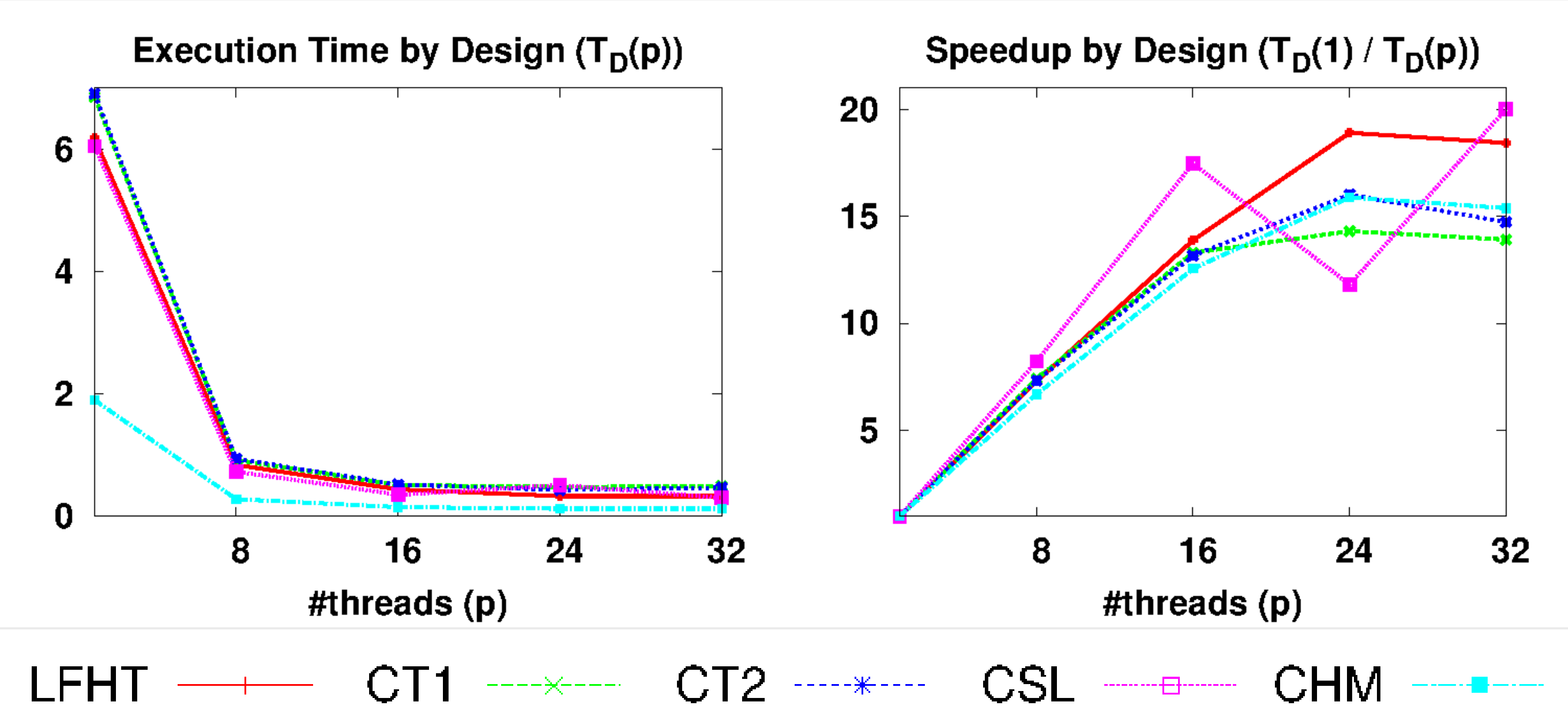
- Comparison in a **32 Core AMD** machine. Threads **insert** different items.



- **LFHT Lock-Free Hash Tries** - **CT1 /CT2 C-Tries Versions (1/2)**
- **CSL Concurrent Skip Lists** - **CHM Concurrent Hash Maps**

Experimental Results - External Framework

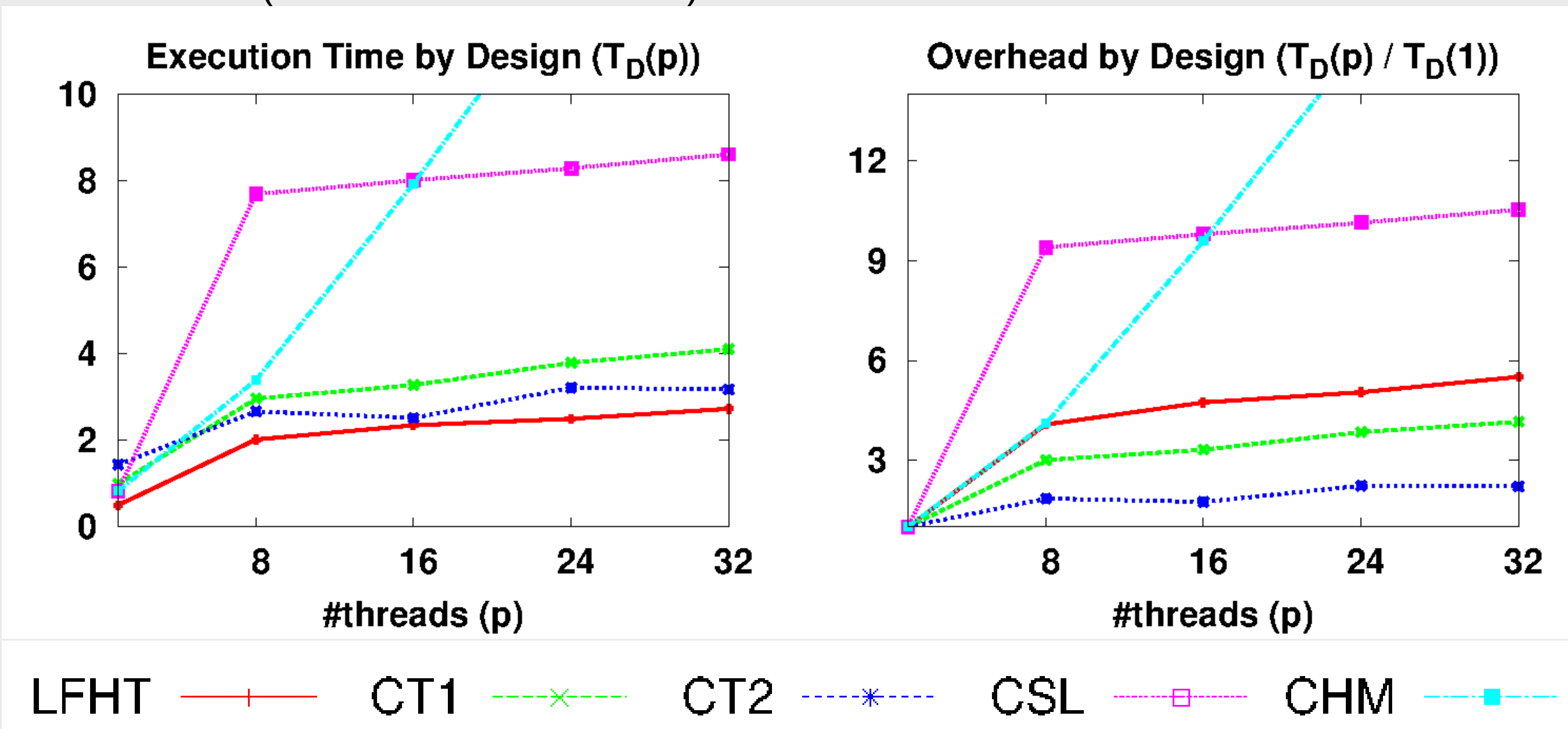
- Comparison in a **32 Core AMD** machine. Threads **lookup** for different items.



- **LFHT Lock-Free Hash Tries** - **CT1 /CT2 C-Tries Versions (1/2)**
- **CSL Concurrent Skip Lists** - **CHM Concurrent Hash Maps**

Experimental Results - External Framework

- Comparison in a **32 Core AMD** machine. All threads **lookup and insert** the same items (**worst case** scenario).



- **LFHT Lock-Free Hash Tries** - **CT1 /CT2 C-Tries Versions (1/2)**
- **CSL Concurrent Skip Lists** - **CHM Concurrent Hash Maps**

Lock-Free Hash Tries - Summary

- We have presented a **second approach** for a **lock-free** trie data structures applied to the multithreaded tabled evaluation of logic programs:
  - Improves the **efficiency** of the **concurrent lookup** and **insert operations** even in **worst case scenarios**.
  - The paper **A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs** discusses the most relevant **implementation details** and **proves the correctness** of the model.

Lock-Free Hash Tries - Summary

- We have presented a **second approach** for a **lock-free** trie data structures applied to the multithreaded tabled evaluation of logic programs:
  - Improves the **efficiency** of the **concurrent lookup** and **insert operations** even in **worst case scenarios**.
  - The paper **A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs** discusses the most relevant **implementation details** and **proves the correctness** of the model.

- Experimental results show that our approach can **effectively** reduce the **execution time** and **scale better**, when increasing the number of threads, than other designs.
  - **Tabling framework**: Our best Lock-Based Tries, Lock-Free Tries and Lock-Free Hash Tries.
  - **External framework**: Concurrent Tries (versions 1 and 2), Concurrent Skip Lists and Concurrent Hash Maps.

Lock-Free Hash Tries - Applications

- Use **Lock-Free Hash Tries** with **Subgoal-Sharing** in the **YapTab-Mt** framework and extend it to support **asynchronous parallelism**.

Lock-Free Hash Tries - Applications

- Use **Lock-Free Hash Tries** with **Subgoal-Sharing** in the **YapTab-Mt** framework and extend it to support **asynchronous parallelism**.
  - The **key idea** is that a **thread does not wait** for **other threads** to **compute** a **sub-problem** ...
  - ... but **is able** to **use the result** of the **sub-problem**, if another **thread** has **already computed** it.

Lock-Free Hash Tries - Applications

- Use **Lock-Free Hash Tries** with **Subgoal-Sharing** in the **YapTab-Mt** framework and extend it to support **asynchronous parallelism**.
  - The **key idea** is that a **thread does not wait** for **other threads** to **compute** a **sub-problem** ...
  - ... but **is able** to **use the result** of the **sub-problem**, if another **thread** has **already computed** it.

- Use the **YapTab-Mt** to **scale the execution** of two well-know **dynamic programming problems** that can be found in many domains:
  - **0-1 Knapsack**: logistics, manufacturing, finance or telecommunications.
  - **Longest Common Subsequence (LCS)**: sequence alignment, which is a fundamental technique for biologists to investigate the similarity between species.

Lock-Free Hash Tries - Applications

- Use **Lock-Free Hash Tries** with **Subgoal-Sharing** in the **YapTab-Mt** framework and extend it to support **asynchronous parallelism**.
  - The **key idea** is that a **thread does not wait** for **other threads** to **compute** a **sub-problem** ...
  - ... but **is able** to **use the result** of the **sub-problem**, if another **thread** has **already computed** it.

- Use the **YapTab-Mt** to **scale the execution** of two well-know **dynamic programming problems** that can be found in many domains:
  - **0-1 Knapsack**: logistics, manufacturing, finance or telecommunications.
  - **Longest Common Subsequence (LCS)**: sequence alignment, which is a fundamental technique for biologists to investigate the similarity between species.

- Compare **parallelization techniques**:
  - **Top-Down** vs **Bottom-Up**.

YapTab-Mt - Advantages

- **Abstraction layer** for the **dynamic programming (tabling)** support is provided with a **single instruction**:
  - **:- table predicate/arity.**
  - Example **:- table knapsack/3.**

YapTab-Mt - Advantages

- **Abstraction layer** for the **dynamic programming (tabling)** support is provided with a **single instruction**:
  - **:- table predicate/arity.**
  - Example **:- table knapsack/3.**

- **Thread API** is **POSIX Threads compliant**:
  - **Management** - creating, joining , yielding, etc.
  - **Monitoring** - statistics, properties, etc.
  - **Synchronization** - mutex creation, statistics, etc.

YapTab-Mt - Advantages

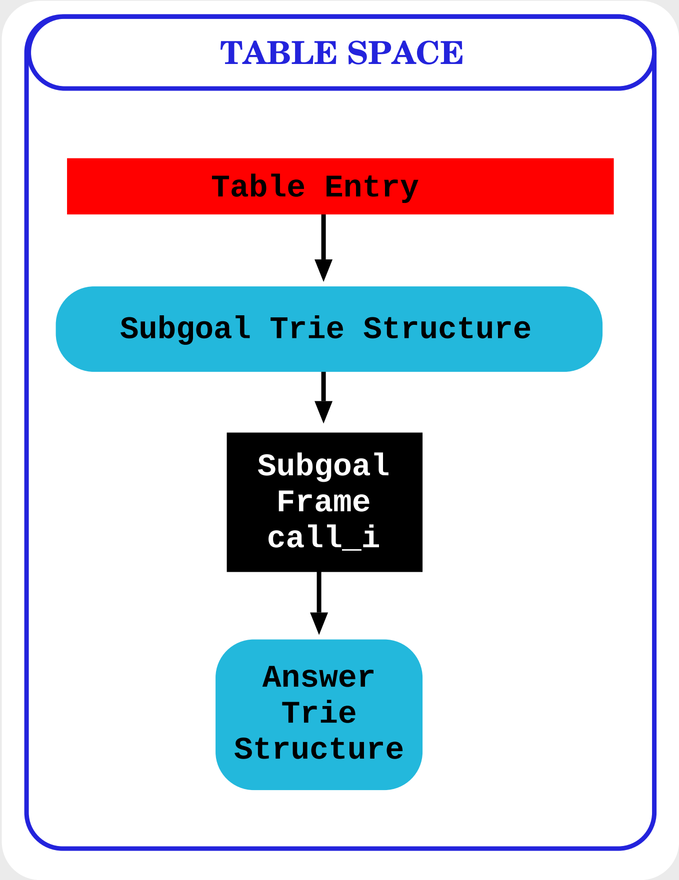
- **Abstraction layer** for the **dynamic programming (tabling)** support is provided with a **single instruction**:
  - **:- table predicate/arity.**
  - Example **:- table knapsack/3.**

- **Thread API** is **POSIX Threads compliant**:
  - **Management** - creating, joining , yielding, etc.
  - **Monitoring** - statistics, properties, etc.
  - **Synchronization** - mutex creation, statistics, etc.

- Write complex **dynamic programming** applications using the **Prolog** programming language.
  - **Procedures** in **Prolog** can be written as **logical specifications**, which are closer to **mathematical notation**.

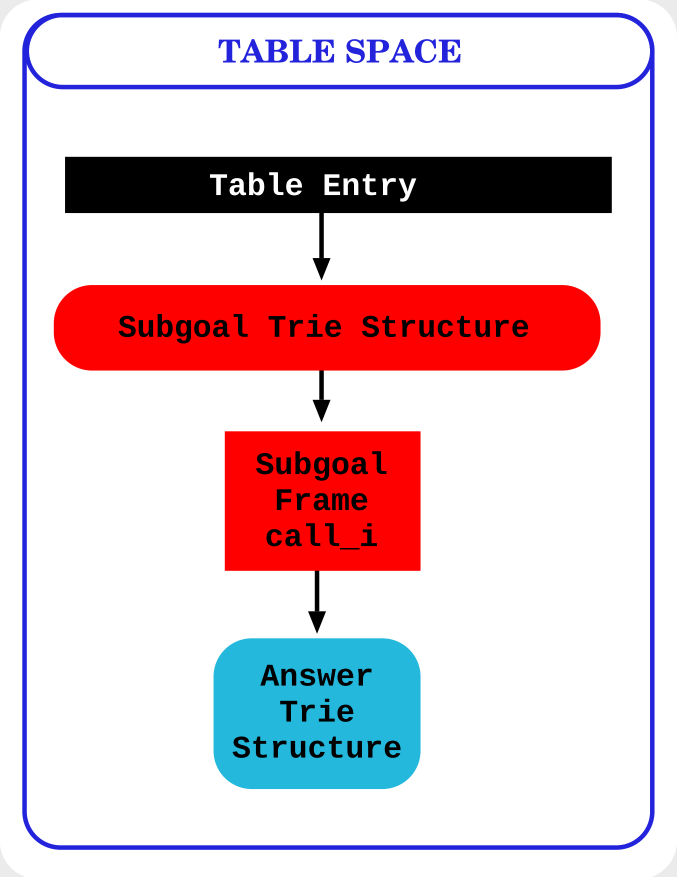
Internal Table Space Architecture

- **Table Entry**: stores generic about the predicates.
  - **table knapsack/3**.



Internal Table Space Architecture

- **Table Entry**: stores generic about the predicates.
  - **table knapsack/3**.
- **Subgoal Trie Structure**: stores the **identifier** of the computations.
  - **knapsack**(**item_i**, **capacity_c**, Profit).



Internal Table Space Architecture

- **Table Entry**: stores generic about the predicates.
  - **table knapsack/3**.
- **Subgoal Trie Structure**: stores the **identifier** of the computations.
  - **knapsack**(**item_i**, **capacity_c**, Profit).
- **Answer Trie Structure**: stores the **answers** of the computations.
  - **knapsack**(item_i, capacity_c, **Profit**).

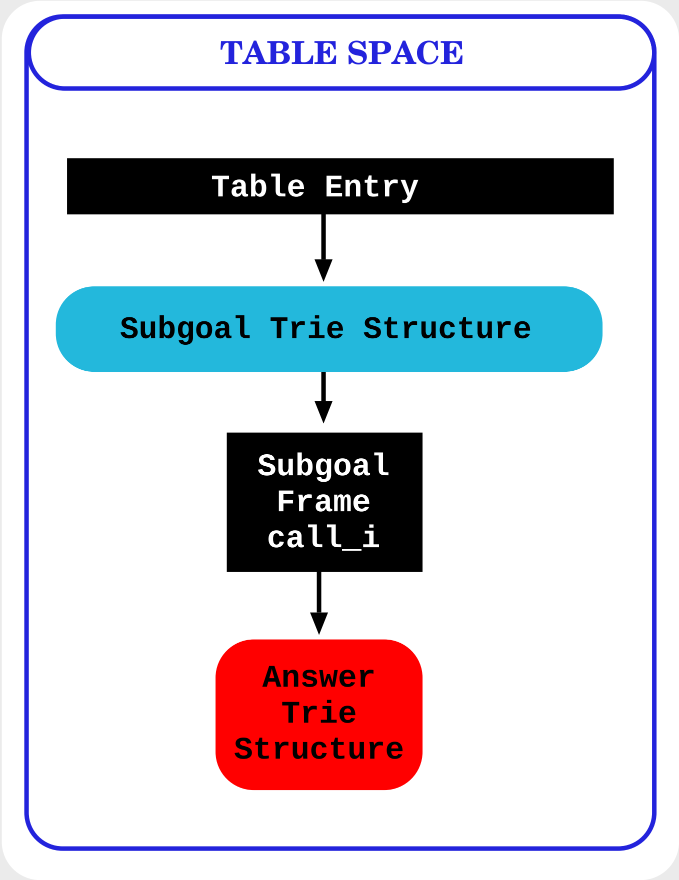


Table Space - Multithreaded Designs

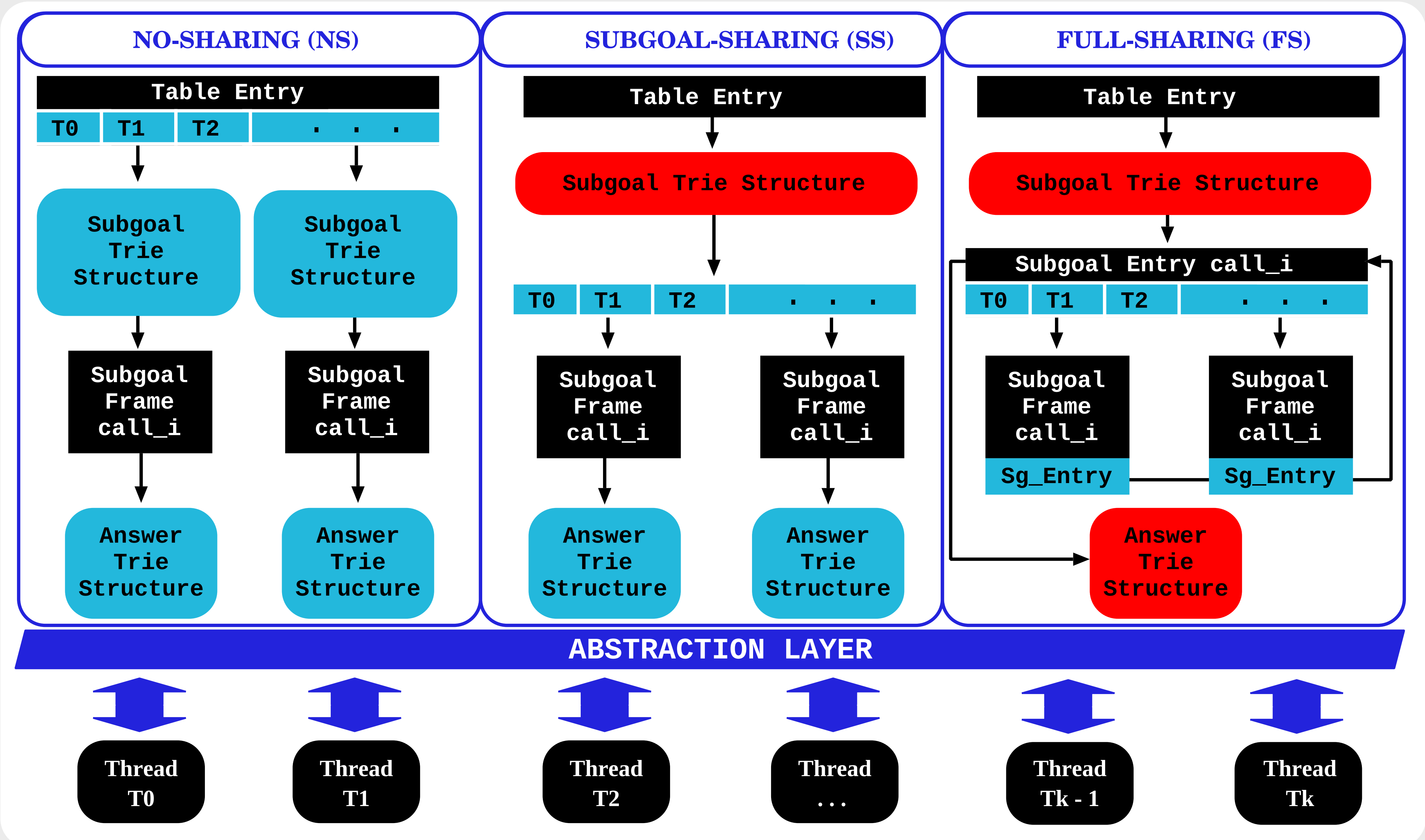


Table Space - Multithreaded Designs

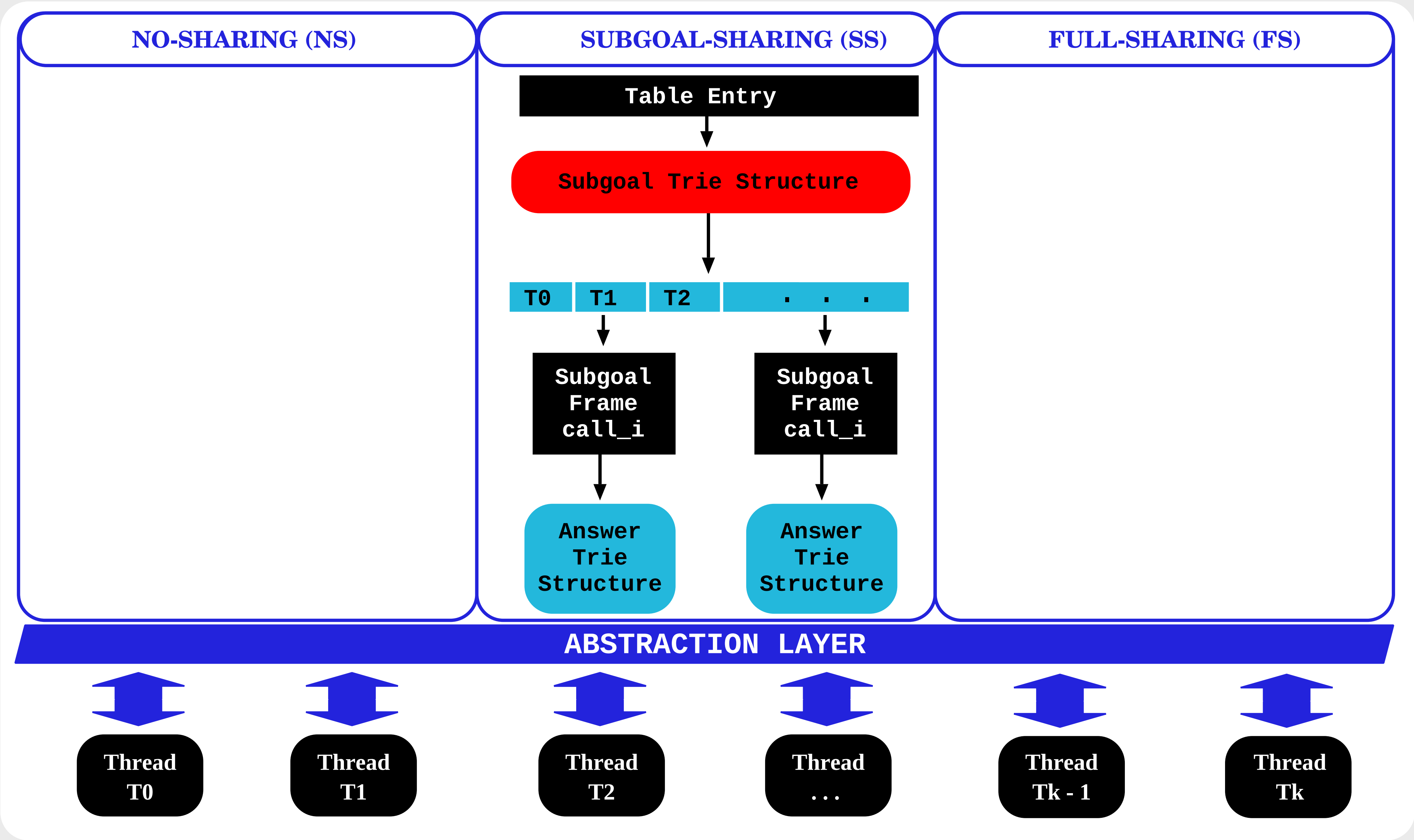


Table Space - Multithreaded Designs

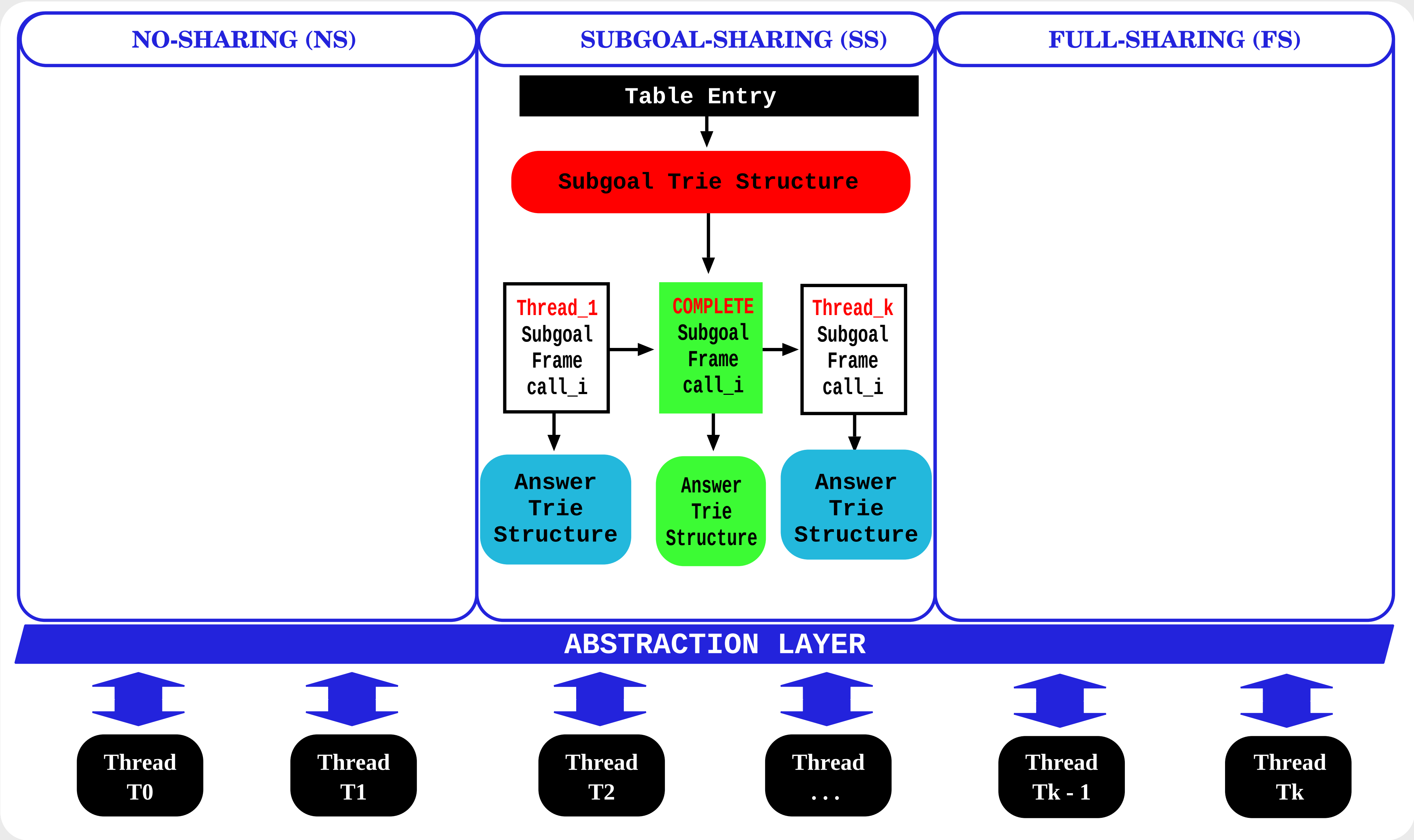
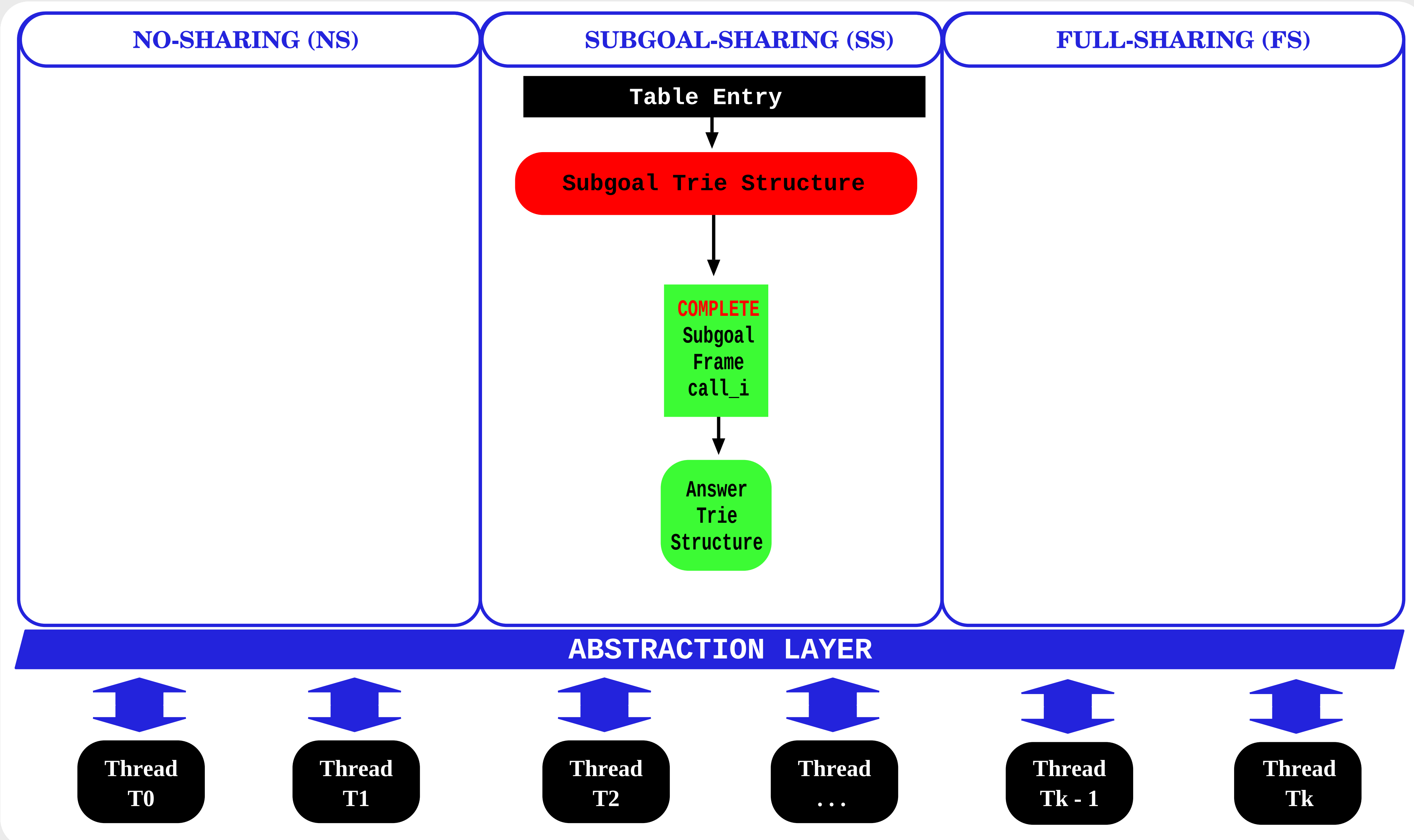
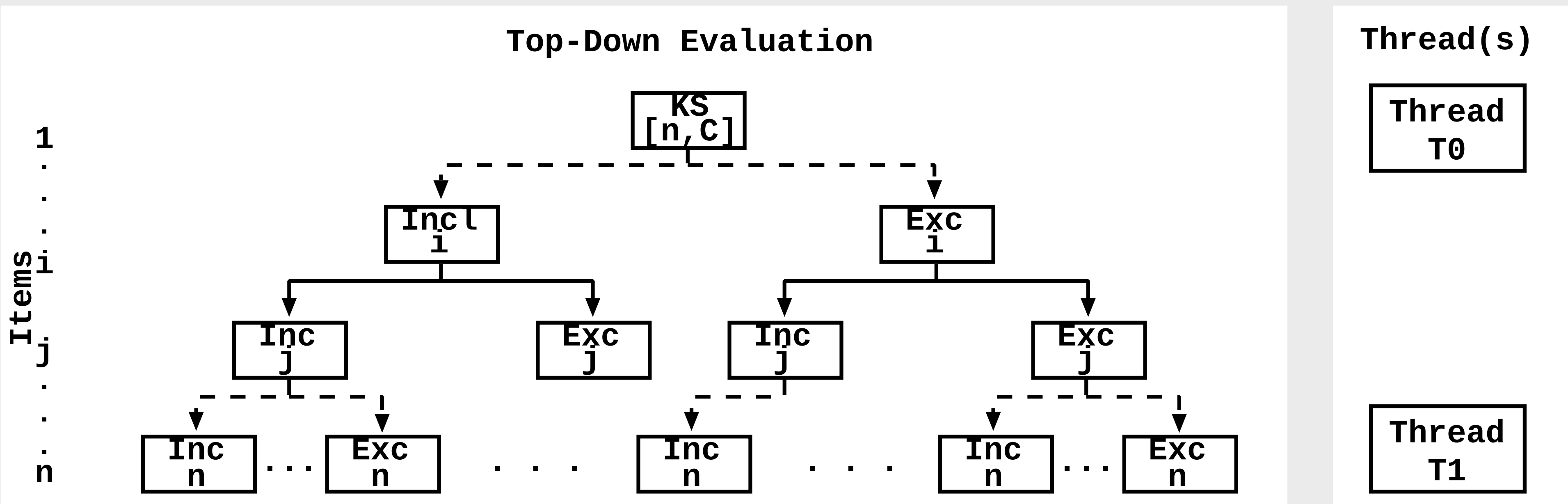


Table Space - Multithreaded Designs



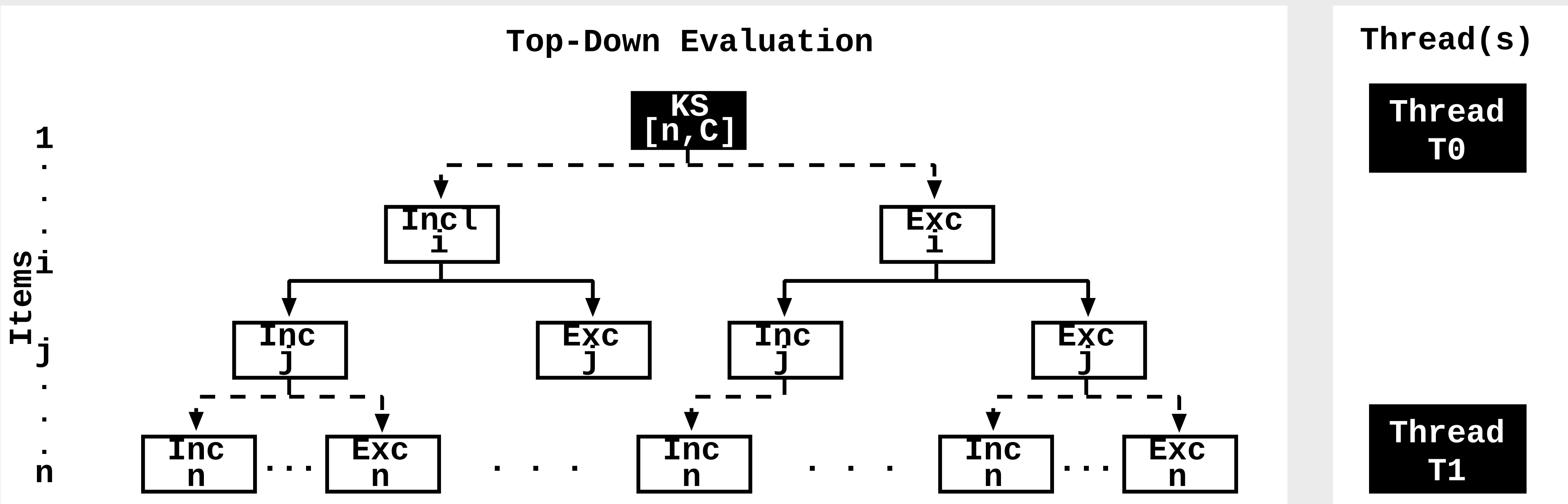
0-1 Knapsack Problem (Top-Down)

- An **item** is **included or excluded** from the **Knapsack** whether it **belongs or not** to the **best solution** of the problem.
- Thread(s) scheduling:
  - Threads **begin** their evaluation in the **top query**.
  - **Disperse threads** through the **evaluation tree** using **random branch orders**.



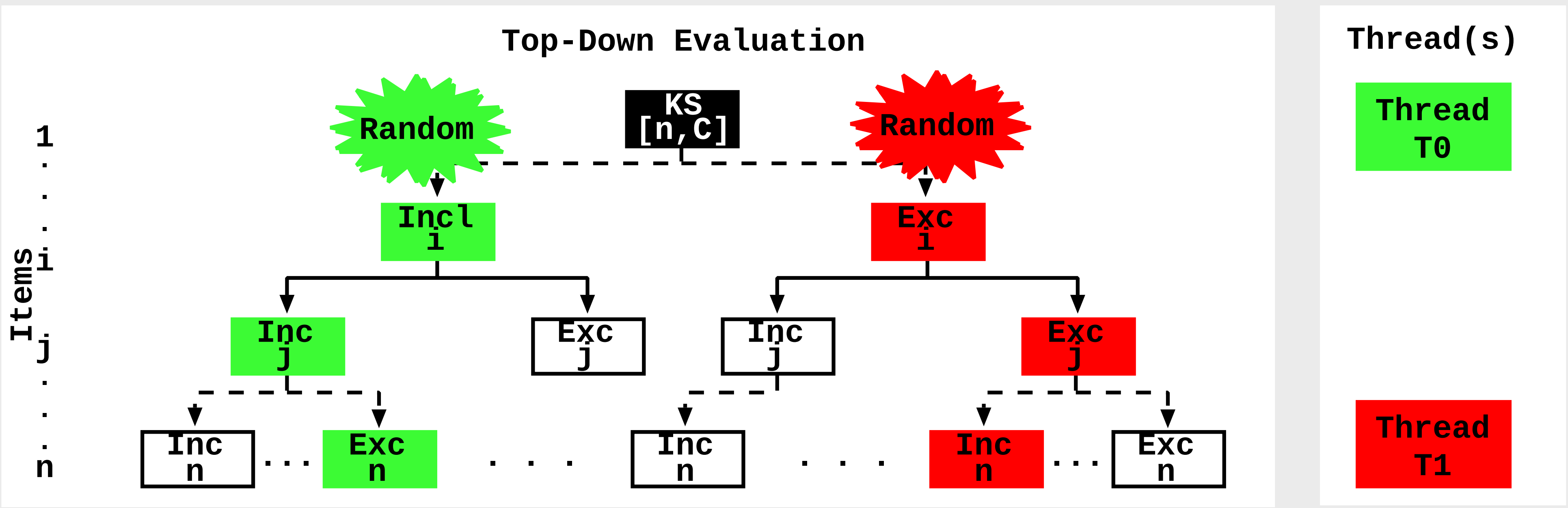
0-1 Knapsack Problem (Top-Down)

- An **item** is **included or excluded** from the **Knapsack** whether it **belongs or not** to the **best solution** of the problem.
- Thread(s) scheduling:
  - Threads **begin** their evaluation in the **top query**.
  - **Disperse threads** through the **evaluation tree** using **random branch orders**.



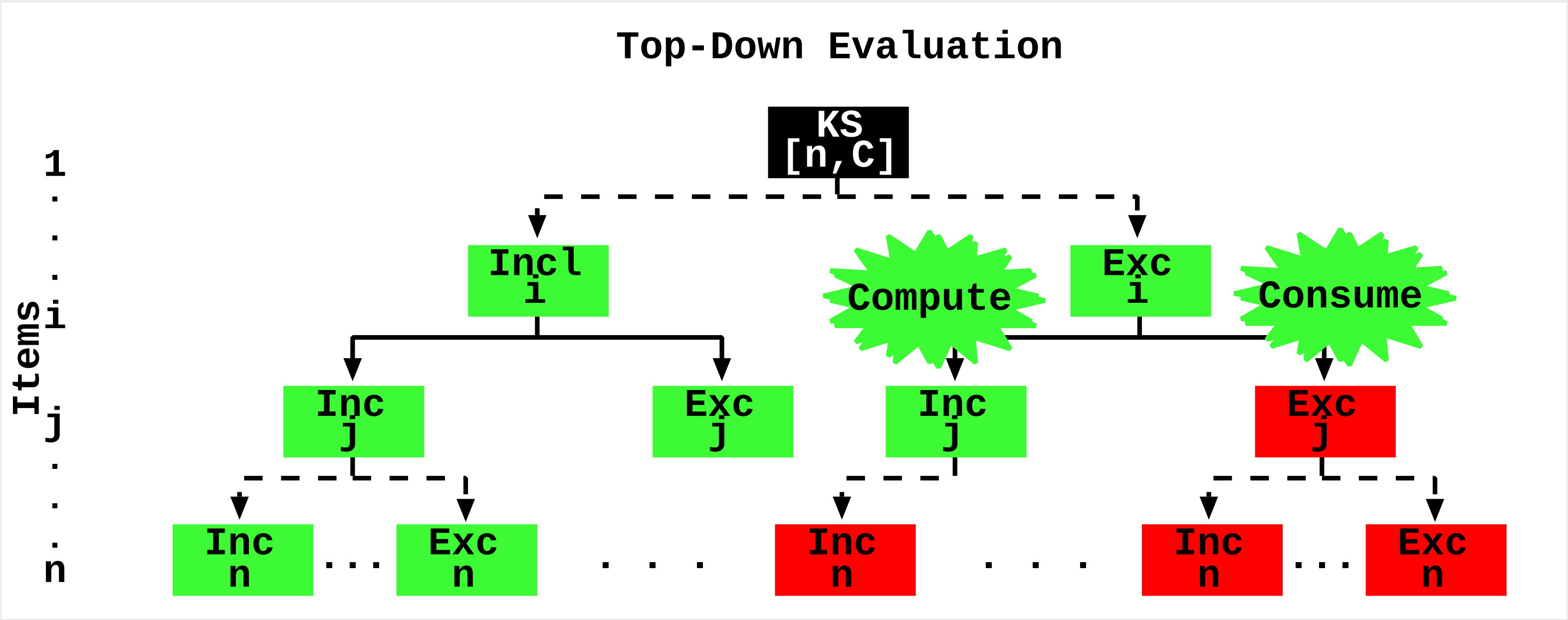
0-1 Knapsack Problem (Top-Down)

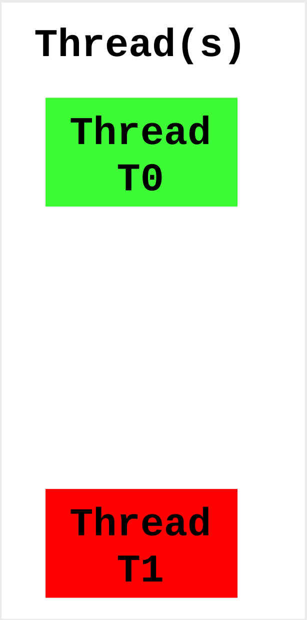
- An **item** is **included or excluded** from the **Knapsack** whether it **belongs or not** to the **best solution** of the problem.
- Thread(s) scheduling:
  - Threads **begin** their evaluation in the **top query**.
  - **Disperse threads** through the **evaluation tree** using **random branch orders**.



0-1 Knapsack Problem (Top-Down)

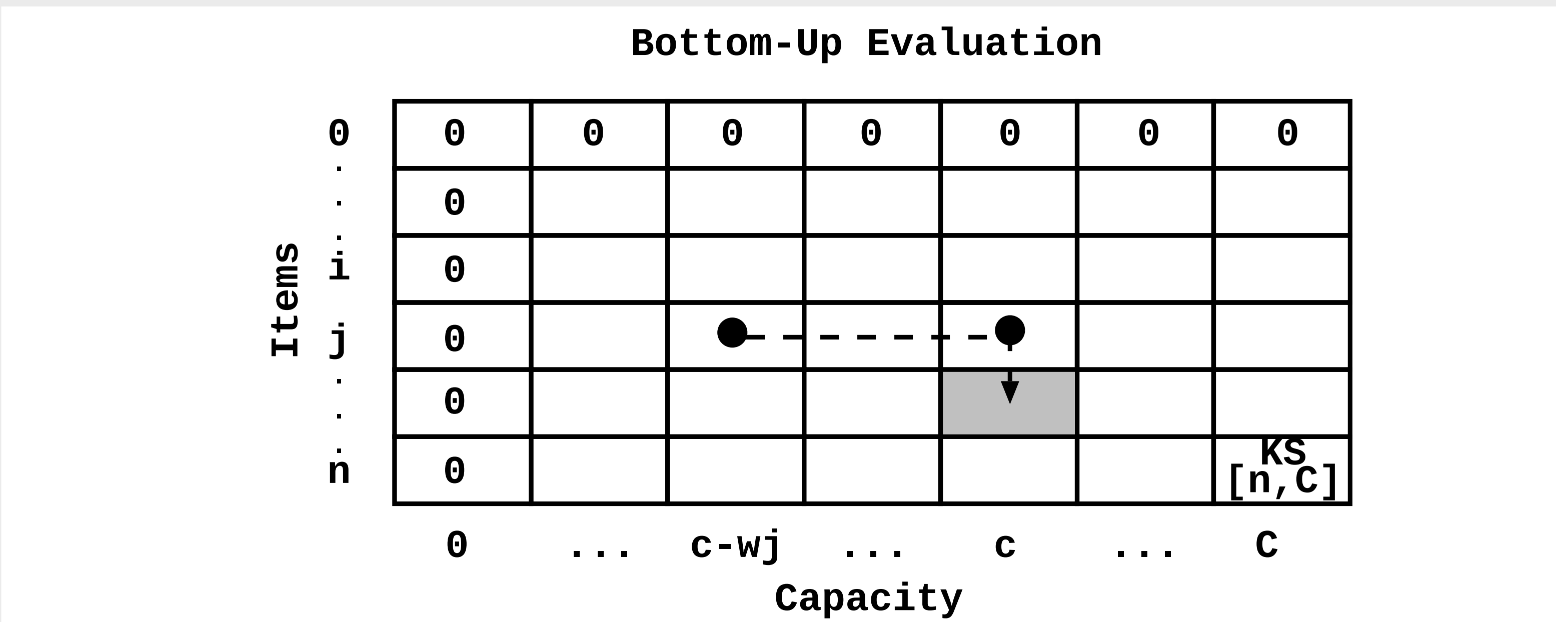
- An **item** is **included or excluded** from the **Knapsack** whether it **belongs or not** to the **best solution** of the problem.
- Thread(s) scheduling:
  - Threads **begin** their evaluation in the **top query**.
  - **Disperse threads** through the **evaluation tree** using **random branch orders**.

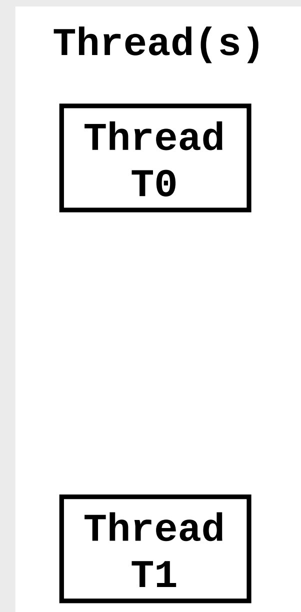




0-1 Knapsack Problem (Bottom-Up)

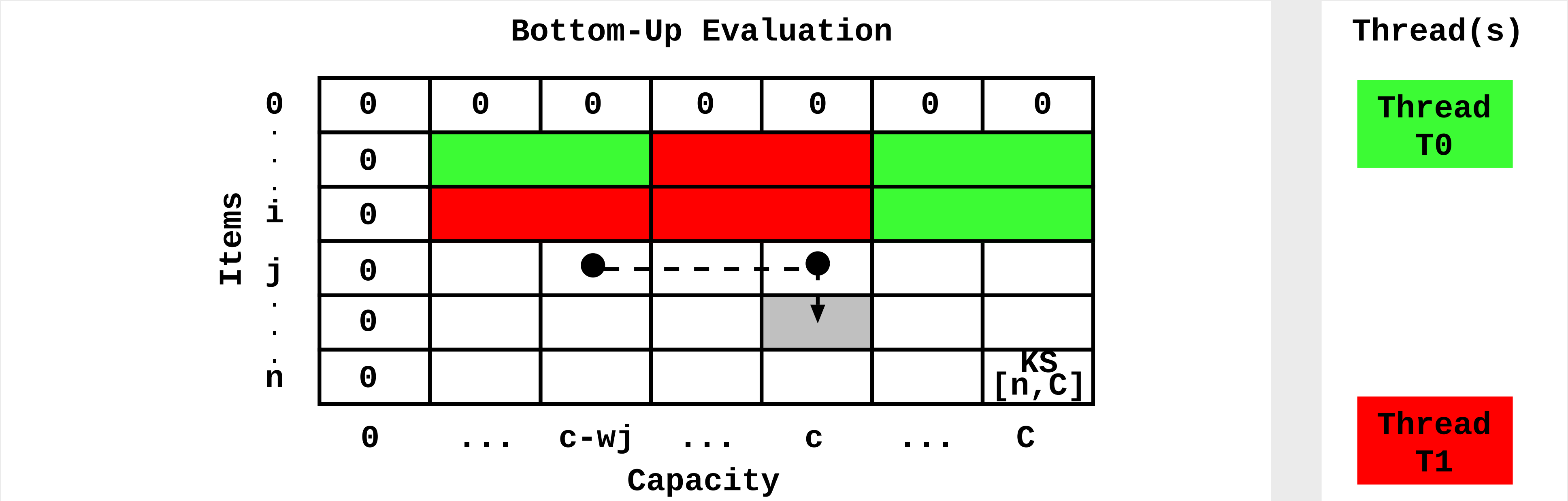
- Evaluate the **combination** of **all items** with **all possible capacities** for the **Knapsack**. After all combinations are evaluated, the **best solution** of the problem has the **items that belong** to the **Knapsack**.
- Thread(s) scheduling:
  - **Divide** the **complete combination** in **smaller chunks** and **evaluate them** in the **threads**.





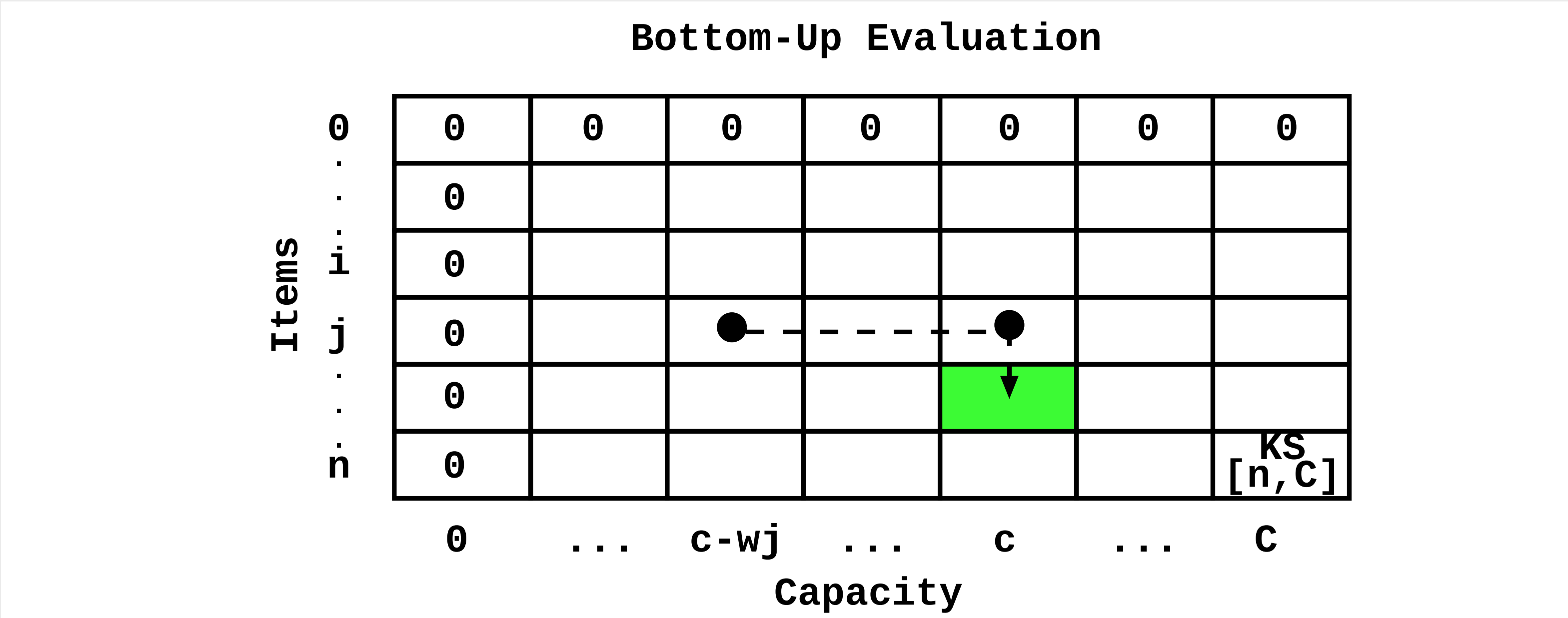
0-1 Knapsack Problem (Bottom-Up)

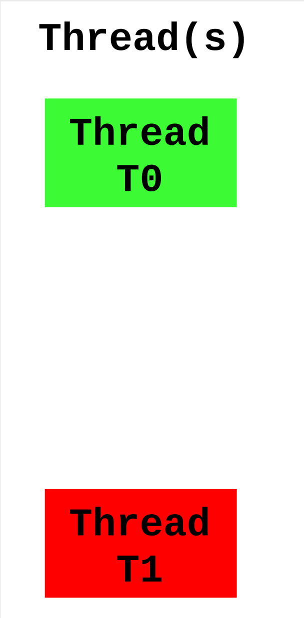
- Evaluate the **combination** of **all items** with **all possible capacities** for the **Knapsack**. After all combinations are evaluated, the **best solution** of the problem has the **items that belong** to the **Knapsack**.
- Thread(s) scheduling:
  - **Divide** the **complete combination** in **smaller chunks** and **evaluate them** in the **threads**.



0-1 Knapsack Problem (Bottom-Up)

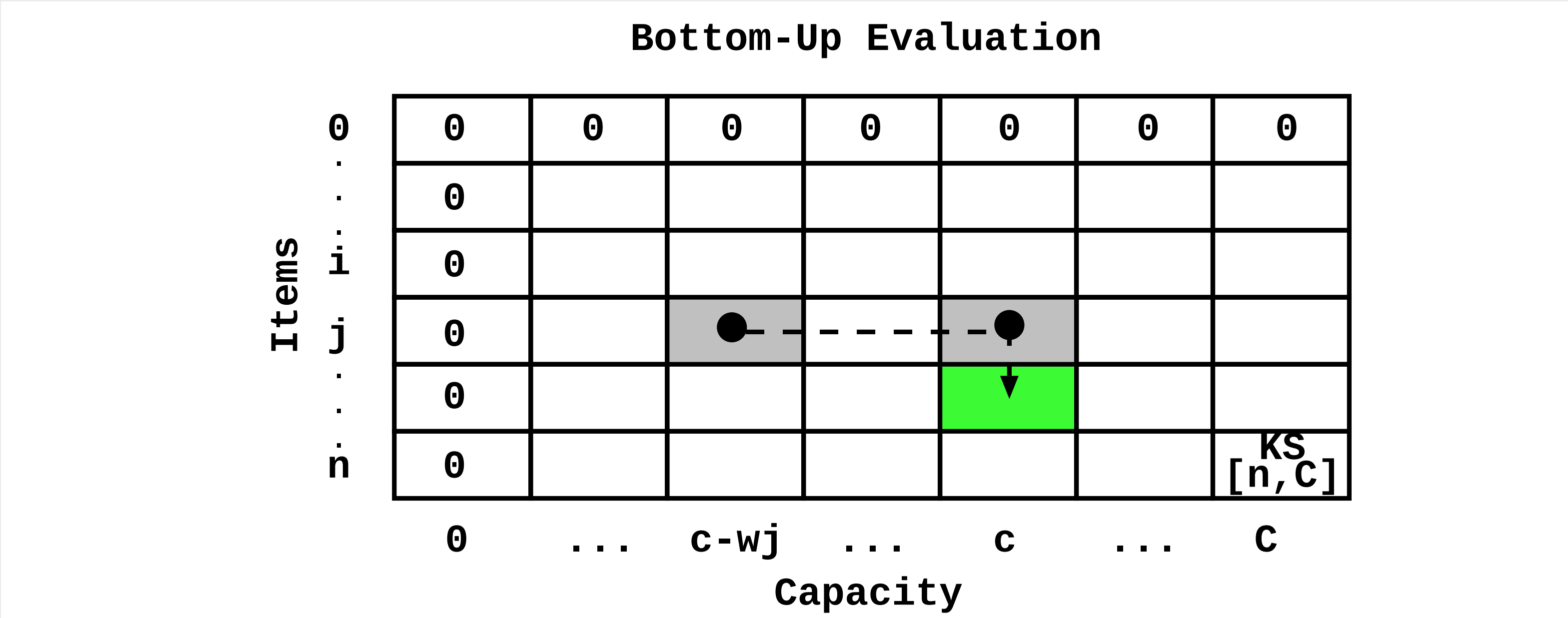
- Evaluate the **combination** of **all items** with **all possible capacities** for the **Knapsack**. After all combinations are evaluated, the **best solution** of the problem has the **items that belong** to the **Knapsack**.
- Thread(s) scheduling:
  - **Divide** the **complete combination** in **smaller chunks** and **evaluate them** in the **threads**.

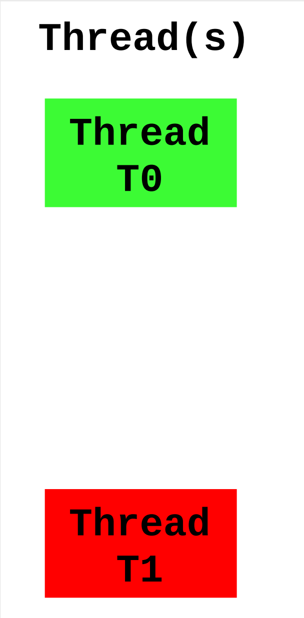




0-1 Knapsack Problem (Bottom-Up)

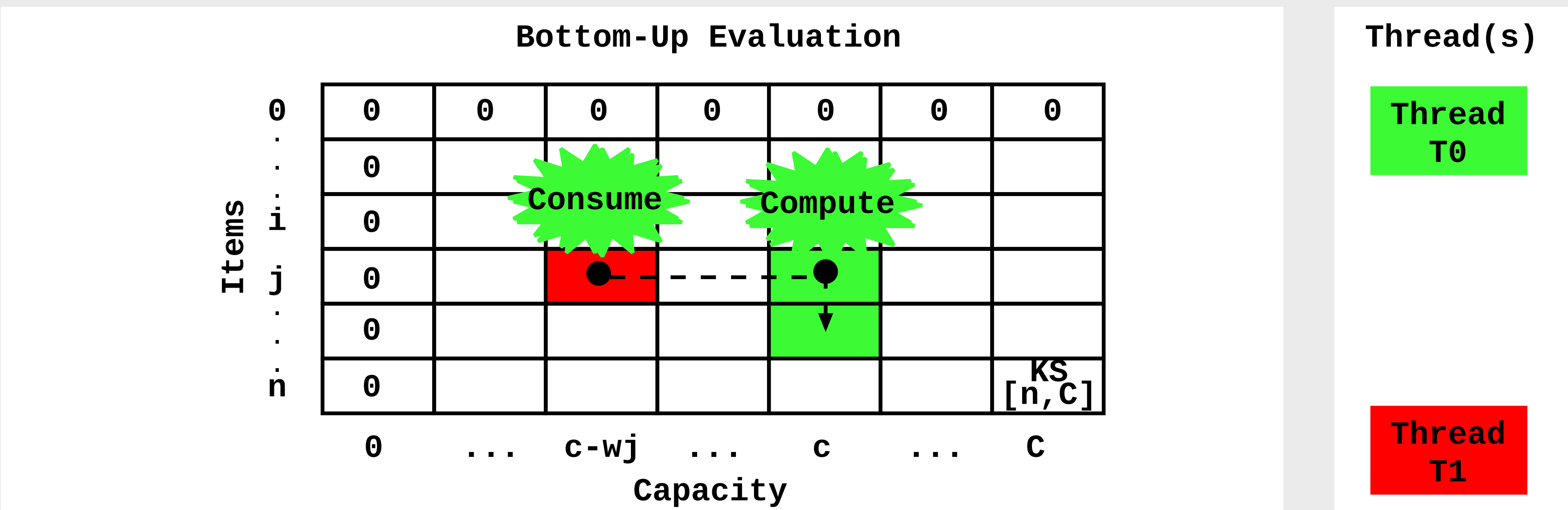
- Evaluate the **combination** of **all items** with **all possible capacities** for the **Knapsack**. After all combinations are evaluated, the **best solution** of the problem has the **items that belong** to the **Knapsack**.
- Thread(s) scheduling:
  - **Divide** the **complete combination** in **smaller chunks** and **evaluate them** in the **threads**.





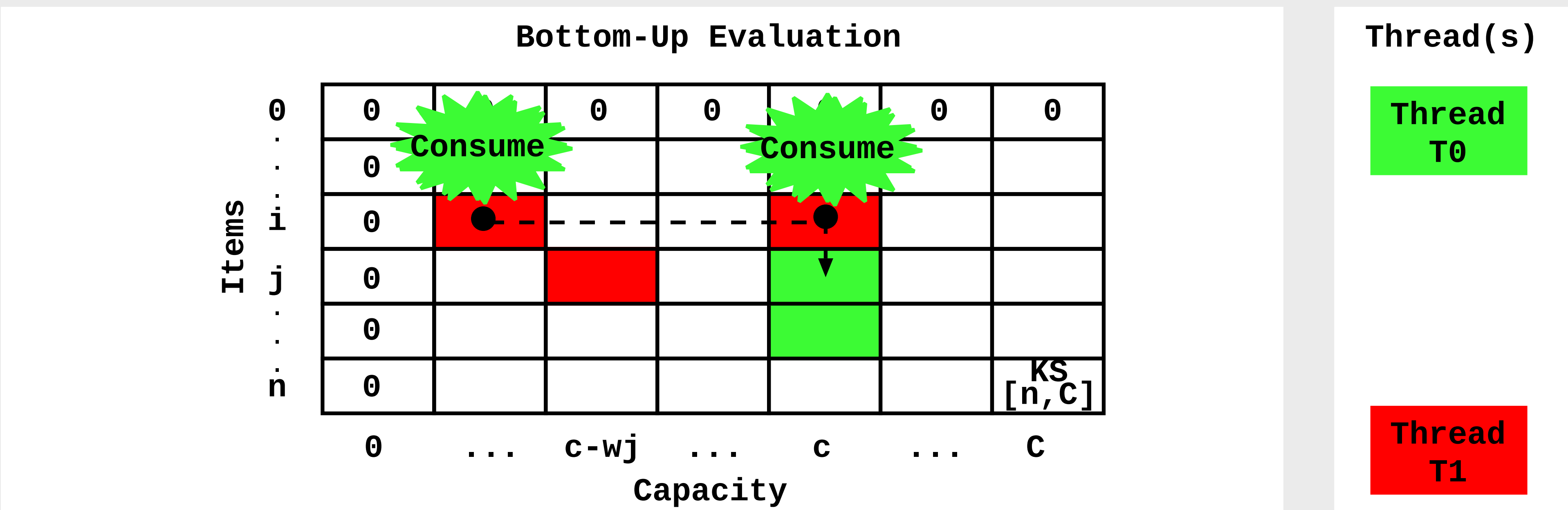
0-1 Knapsack Problem (Bottom-Up)

- Evaluate the **combination** of **all items** with **all possible capacities** for the **Knapsack**. After all combinations are evaluated, the **best solution** of the problem has the **items that belong** to the **Knapsack**.
- Thread(s) scheduling:
  - **Divide** the **complete combination** in **smaller chunks** and **evaluate them** in the **threads**.



0-1 Knapsack Problem (Bottom-Up)

- Evaluate the **combination** of **all items** with **all possible capacities** for the **Knapsack**. After all combinations are evaluated, the **best solution** of the problem has the **items that belong** to the **Knapsack**.
- Thread(s) scheduling:
  - **Divide** the **complete combination** in **smaller chunks** and **evaluate them** in the **threads**.



Experimental Results - 0-1 Knapsack Problem

System/Dataset		# Threads (p) Time (T_1) 1	Speedup (T_1/T_p) 8	16	24	32	Best Time
Top-Down Approaches							
YAP_{TD_1}	D_{10}	18,319	1.96	**2.10**	2.01	1.89	8,723
	D_{30}	17,664	3.41	**3.96**	3.83	3.62	4,461
	D_{50}	17,828	4.72	6.12	**6.21**	6.07	2,871
YAP_{TD_2}	D_{10}	23,816	6.78	11.95	14.81	**16.79**	1,418
	D_{30}	25,049	7.39	13.63	16.85	**19.35**	1,295
	D_{50}	24,866	7.38	13.67	16.78	**19.23**	1,293
Bottom-Up Approaches							
YAP_{BU}	D_{10}	17,054	7.25	13.32	17.12	**19.60**	0,870
	D_{30}	17,005	7.22	13.47	17.29	**19.64**	0,866
	D_{50}	16,550	7.16	13.29	17.04	**19.60**	0,844
XSB_{BU}	D_{10}	37,338	**0.81**	0.79	0.73	0.54	37,338
	D_{30}	38,245	**0.82**	0.75	0.75	0.56	38,245
	D_{50}	39,100	**0.82**	0.79	0.73	0.54	39,100

Experimental Results - LCS Problem

System/Dataset		Time (T_1)	# Threads (p) Speedup (T_1/T_p)				Best
		1	8	16	24	32	Time
Top-Down Approaches							
YAP_{TD_1}	D_{10}	30,708	**1.53**	1.45	1.40	1.29	20,071
	D_{30}	30,817	**1.53**	1.46	1.38	1.28	20,142
	D_{50}	30,707	**1.52**	1.44	1.39	1.27	20,202
YAP_{TD_2}	D_{10}	42,556	7.25	13.13	16.26	**18.32**	2,323
	D_{30}	42,511	7.21	13.24	16.19	**18.34**	2,318
	D_{50}	42,631	7.21	13.15	16.27	**18.33**	2,326
Bottom-Up Approaches							
YAP_{BU}	D_{10}	27,253	6.97	10.78	14.88	**17.91**	1,522
	D_{30}	27,045	6.88	11.20	14.74	**17.92**	1,509
	D_{50}	27,102	6.97	11.91	14.51	**18.07**	1,500
XSB_{BU}	D_{10}	68,255	n.a.	n.a.	n.a.	n.a.	68,255
	D_{30}	69,700	n.a.	n.a.	n.a.	n.a.	69,700
	D_{50}	70,100	n.a.	n.a.	n.a.	n.a.	70,100

Conclusions

- We have **showed two lock-free approaches** for the implementation of **concurrent Tries** and **compared**:
  - Both of them against our lock-based implementations.
  - The **Lock-Free Hash Tries** version against **other lock-free implementations**.

Conclusions

- We have **showed two lock-free approaches** for the implementation of **concurrent Tries** and **compared**:
  - Both of them against our lock-based implementations.
  - The **Lock-Free Hash Tries** version against **other lock-free implementations**.

- Used the **Lock-Free Hash Tries** with **Subgoal-Sharing** in the YapTab-Mt framework and extend it with **asynchronous parallelism**.
  - The **0-1 Knapsack** and the **Longest Common Subsequence** problems are two well-know **dynamic programming problems**.

Conclusions

- We have **showed two lock-free approaches** for the implementation of **concurrent Tries** and **compared**:
  - Both of them against our lock-based implementations.
  - The **Lock-Free Hash Tries** version against **other lock-free implementations**.

- Used the **Lock-Free Hash Tries** with **Subgoal-Sharing** in the YapTab-Mt framework and extend it with **asynchronous parallelism**.
  - The **0-1 Knapsack** and the **Longest Common Subsequence** problems are two well-know **dynamic programming problems**.
  - We have **discussed** how we were able to **scale the execution** by taking advantage of the **YapTap-Mt** framework.
    - **Top-Down** vs **Bottom-Up**.

Conclusions

- We have **showed two lock-free approaches** for the implementation of **concurrent Tries** and **compared**:
  - Both of them against our lock-based implementations.
  - The **Lock-Free Hash Tries** version against **other lock-free implementations**.

- Used the **Lock-Free Hash Tries** with **Subgoal-Sharing** in the YapTab-Mt framework and extend it with **asynchronous parallelism**.
  - The **0-1 Knapsack** and the **Longest Common Subsequence** problems are two well-know **dynamic programming problems**.
  - We have **discussed** how we were able to **scale the execution** by taking advantage of the **YapTap-Mt** framework.
    - **Top-Down** vs **Bottom-Up**.

  - The paper **On Scaling Dynamic Programming Problems with a Multithreaded Tabling System** shows the **Prolog code** and other **interesting details**.

Thank You !!!

Miguel Areias and Ricardo Rocha
CRACS & INESC-TEC LA
University of Porto, Portugal
miguel-areias@dcc.fc.up.pt *ricroc@dcc.fc.up.pt*

Yap Prolog: *http://www.dcc.fc.up.pt/~vsc/Yap*

Projects SIBILA: *http://cracs.fc.up.pt/*

FCT Grant: *SFRH/BD/69673/2010*

COMPETE
PROGRAMA OPERACIONAL FACTORES DE COMPETITIVIDADE
QREN QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL PORTUGAL 2007.2013
UNIÃO EUROPEIA Fundo Europeu de Desenvolvimento Regional
FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA