香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Rapids @ HKUST

# Scaling Up Subgraph Query Processing with Efficient Subgraph Matching

***Shixuan Sun*** *and Qiong Luo**

*The Hong Kong University of Science and Technology*

# Subgraph Isomorphism

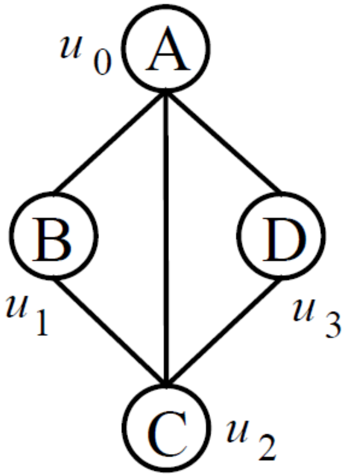
Given graphs $g = (V, E, L)$ and $g' = (V', E', L')$, a subgraph isomorphism from $g$ to $g'$ is an injective function $\varphi: V \to V'$

that satisfies:

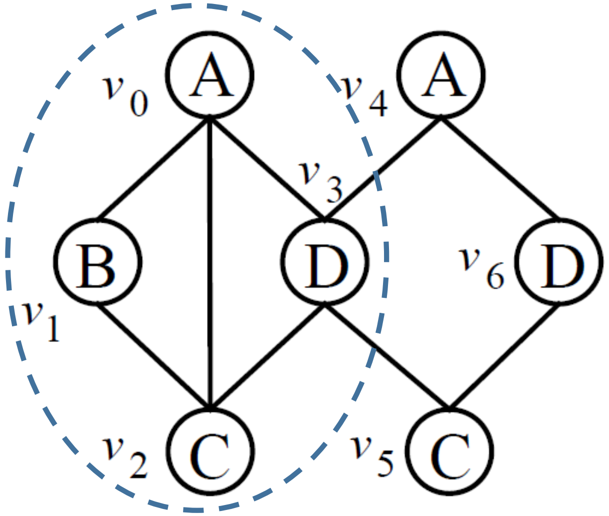
(1) $\forall u \in V, L(u) = L'(\varphi(u))$;

(2) $\forall e(u, u') \in E, e(\varphi(u), \varphi(u')) \in E'$.

# Example of Subgraph Isomorphism




Graph $g$.




Graph $g'$.

$$\varphi_0 = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3)\}$$

A Subgraph Isomorphism from $g$ to $g'$.

# Problem Definition

Given a graph database $D = \{G_1, G_2, \dots, G_n\}$ and a query graph $q$, a subgraph query finds all data graphs in $D$ that contain $q$.

Applications:

- Computer-aided design;
- Protein interaction relationship retrieval.

# A Naïve Solution

Loop over each data graph $G$ in graph database $D$ to test whether $G$ contains the query graph $q$.

# A Naïve Solution

Loop over each data graph $G$ in graph database $D$ to test whether $G$ contains the query graph $q$.

- ☹ The subgraph isomorphism problem is NP-complete.
- ☹ Perform $|D|$ subgraph isomorphism tests.

# IFV Procedure

Researchers proposed the indexing-filtering-verification (IFV) procedure.

- ☺ Reduce the number of subgraph isomorphism tests.

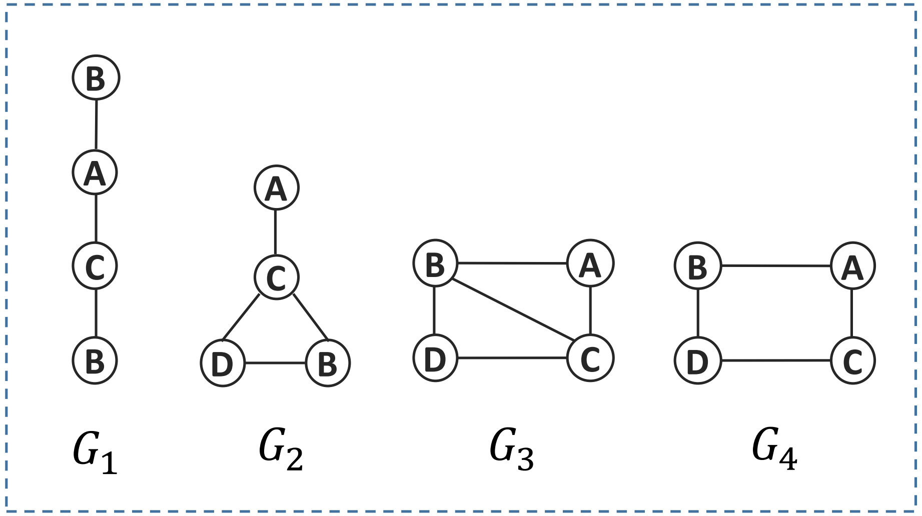
| **Algorithm** | **Feature Extraction** | **Feature Structure** | **Index Storage** |
|---|---|---|---|
| GraphGrep [30] | Enumeration | Path | Memory |
| GraphGrepSX [2] | Enumeration | Path | Memory |
| Grapes [10] | Enumeration | Path | Memory |
| SING [7] | Enumeration | Path | Memory |
| CT-Index [20] | Enumeration | Tree/Cycle | Memory |
| GDIndex [35] | Enumeration | Graph | Memory |
| GCode [43] | Enumeration | Graph | Memory |
| SwiftIndex [28] | Mining | Tree | Memory |
| TreePi [40] | Mining | Tree | Memory |
| Tree+Delta [42] | Mining | Tree/Graph | Memory |
| CP-Index [36] | Mining | Graph | Memory |
| gIndex [37] | Mining | Graph | Memory |
| FG-Index [4] | Mining | Graph | Memory/Disk |
| FG*-Index [3] | Mining | Graph | Memory/Disk |
| Lindex+ [38] | Mining | Graph | Memory/Disk |

# Indexing

Build an index on data graphs.

- Keys are features.
- Values are data graphs.

# Example of Indexing




Graph Database $D$.

Build $I$ by exhaustively enumerating all paths the length of which is 1.

| Feature | Data Graph ID |
|---|---|
| (A, B) | $G_1, G_3, G_4$ |
| (A, C) | $G_1, G_2, G_3, G_4$ |
| (B, C) | $G_1, G_2, G_3$ |
| (B, D) | $G_2, G_3, G_4$ |
| (C, D) | $G_2, G_3, G_4$ |

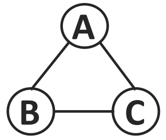
Index $I$.

# Index-Based Filtering

Filter data graphs based on the index.

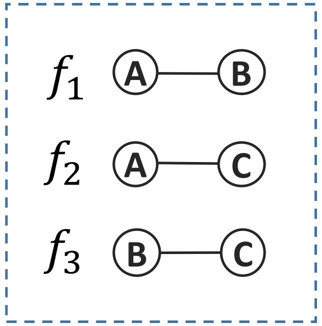
1. Decompose $q$ into a collection $F(q)$ of features.
2. Generate a set $C(q)$ of candidate data graphs, each of which contains $F(q)$.

# Example of Index-Based Filtering




Query Graph $q$.

Decompose




Feature Set $F(q)$.

Lookup

| Feature | Data Graph ID |
|---|---|
| (A, B) | $G_1, G_3, G_4$ |
| (A, C) | $G_1, G_2, G_3, G_4$ |
| (B, C) | $G_1, G_2, G_3$ |
| (B, D) | $G_2, G_3, G_4$ |
| (C, D) | $G_2, G_3, G_4$ |

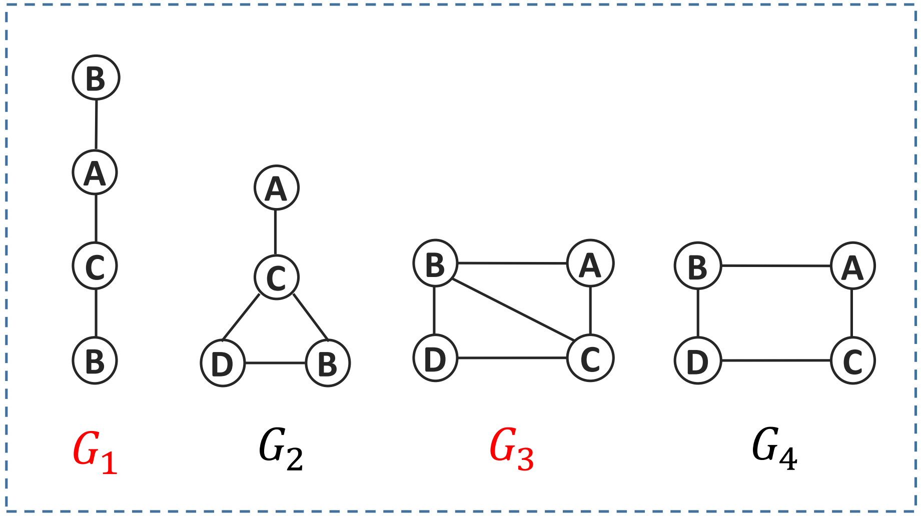
Index $I$.

Intersect

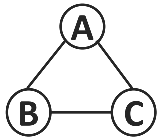
$C(q) = \{G_1, G_3\}$

Candidate Set.

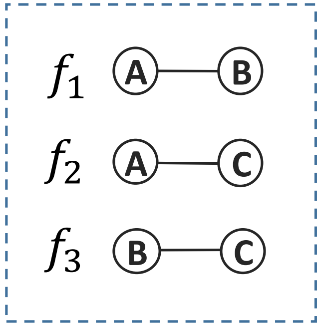
# Example of Index-Based Filtering




Graph Database $D$.



Query Graph $q$.




Feature Set $F(q)$.

# Verification

Verify whether each candidate graph contains the query graph.

- Consider the candidate set $C(q)$ instead of the entire graph database $D$.

# Drawbacks of IFV

☹ Fail to build indices on large graph databases.

F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. In PVLDB, 2015.

# Drawbacks of IFV

- ☹ Fail to build indices on large graph databases.
- ☹ Unsuitable for graphs that change frequently.

# Drawbacks of IFV

- ☹ Fail to build indices on large graph databases.
- ☹ Unsuitable for graphs that change frequently.
- ☹ Slow verification algorithms.

# Subgraph Matching

Given a data graph $G$ and a query graph $q$, subgraph matching finds all subgraph isomorphisms from $q$ to $G$.

# Opportunities from Subgraph Matching

- ☺ The latest subgraph matching algorithms can speed up the verification step in subgraph queries.
- ☺ The preprocessing-enumeration methodology of latest subgraph matching algorithms can also be applied for subgraph queries.

# Preprocessing of Subgraph Matching

Given $q$ and $G$, a candidate vertex set $\Phi(u)$ of $u \in V(q)$ is complete if $\Phi(u)$ satisfies: if the mapping $(u, v)$ exists in a subgraph isomorphism from $q$ to $G$ where $v \in V(G)$, then $v \in \Phi(u)$.

# Preprocessing of Subgraph Matching

Given $q$ and $G$, a candidate vertex set $\Phi(u)$ of $u \in V(q)$ is complete if $\Phi(u)$ satisfies: if the mapping $(u, v)$ exists in a subgraph isomorphism from $q$ to $G$ where $v \in V(G)$, then $v \in \Phi(u)$.

Preprocessing aims to minimize $\Phi(u)$ without breaking its completeness.

F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In SIGMOD, 2016.

# Observation

Given $q$ and $G$, $\forall u \in V(q), \Phi(u)$ is complete. If $\exists u \in V(q), \Phi(u) = \emptyset$, then $G$ does not contain $q$.

# vcFV Procedure

We design the vertex-connectivity based filtering-verification (vcFV) procedure.

- Identify candidate graphs based on candidate vertex sets.

# vcFV Procedure

**Input**: a query graph $q$ and a graph database $\mathcal{D}$

**Output**: an answer set $\mathcal{A}(q)$ keeping all data graphs in $\mathcal{D}$ that contain $q$

**begin**

    $\mathcal{A}(q) \leftarrow \emptyset$;

    **foreach** $G \in \mathcal{D}$ **do**

        $\Phi \leftarrow \texttt{Filter}(q, G)$;

        **if** $\forall u \in V(q), \Phi(u) \neq \emptyset$ **then**

            **if** $\texttt{Verify}(q, G, \Phi)$ *is true* **then**

                $\mathcal{A}(q) \leftarrow \mathcal{A}(q) \cup \{G\}$;

    **return** $\mathcal{A}(q)$;

Algo 1. The vcFV Procedure.

# vcFV Procedure

**Input**: a query graph $q$ and a graph database $\mathcal{D}$
**Output**: an answer set $\mathcal{A}(q)$ keeping all data graphs in $\mathcal{D}$ that contain $q$

```
begin
    A(q) ← ∅;
    foreach G ∈ D do
        Φ ← Filter(q, G);
        if ∀u ∈ V(q), Φ(u) ≠ ∅ then
            if Verify(q, G, Φ) is true then
                A(q) ← A(q) ∪ {G};
    return A(q);
```

① Use the preprocessing phase of latest subgraph matching algorithms as the filtering function.

Algo 1. The vcFV Procedure.

# vcFV Procedure

**Input**: a query graph $q$ and a graph database $\mathcal{D}$

**Output**: an answer set $\mathcal{A}(q)$ keeping all data graphs in $\mathcal{D}$ that contain $q$

**begin**
    $\mathcal{A}(q) \leftarrow \emptyset$;
    **foreach** $G \in \mathcal{D}$ **do**
        $\boxed{\Phi \leftarrow \texttt{Filter}(q, G);}$
        **if** $\forall u \in V(q), \Phi(u) \neq \emptyset$ **then**
            **if** $\boxed{\texttt{Verify}(q, G, \Phi)\ \textit{is true}}$ **then**
                $\mathcal{A}(q) \leftarrow \mathcal{A}(q) \cup \{G\}$;
    **return** $\mathcal{A}(q)$;

① Use the preprocessing phase of latest subgraph matching algorithms as the filtering function.

② Modify the enumeration phase of latest subgraph matching algorithms as the verification function.

Algo 1. The vcFV Procedure.

# Competing Algorithms

| Category | Algorithm | Indexing | Filtering | Verification |
|---|---|---|---|---|
| **IFV** | CT-Index | Hashset | Index | VF2 |
| | Grapes | Trie | Index | VF2 |
| | GGSX | Suffix tree | Index | VF2 |
| **vcFV** | CFL | N/A | Preprocesing of CFL | Enumeration of CFL |
| | GraphQL | N/A | Preprocesing of GraphQL | Enumeration of GraphQL |
| | CFQL | N/A | Preprocesing of CFL | Enumeration of GraphQL |
| **IvcFV** | vcGrapes | Trie | Index and preprocesing of CFL | Enumeration of GraphQL |
| | vcGGSX | Suffix tree | Index and preprocesing of CFL | Enumeration of GraphQL |

A summary of competing algorithms.

- CT-Index, GGSX and Grapes are the top-performing IFV algorithms in the previous performance study.
- CFL and GraphQL are the leading subgraph matching algorithms.
- IvcFV algorithms are obtained by integrating IFV algorithms with vcFV algorithms.

# Experimental Setup

## Real-world Datasets:

| | AIDS | PDBS | PCM | PPI |
|---|---|---|---|---|
| **#graphs** | 40,000 | 600 | 200 | 20 |
| **#labels** | 62 | 10 | 21 | 46 |
| **#vertices per graph** | 45 | 2,939 | 377 | 4,942 |
| **#edges per graph** | 46.95 | 3,064 | 4,340 | 26,667 |
| **degree per graph** | 2.09 | 2.06 | 23.01 | 10.87 |
| **#labels per graph** | 4.4 | 6.4 | 18.9 | 28.5 |

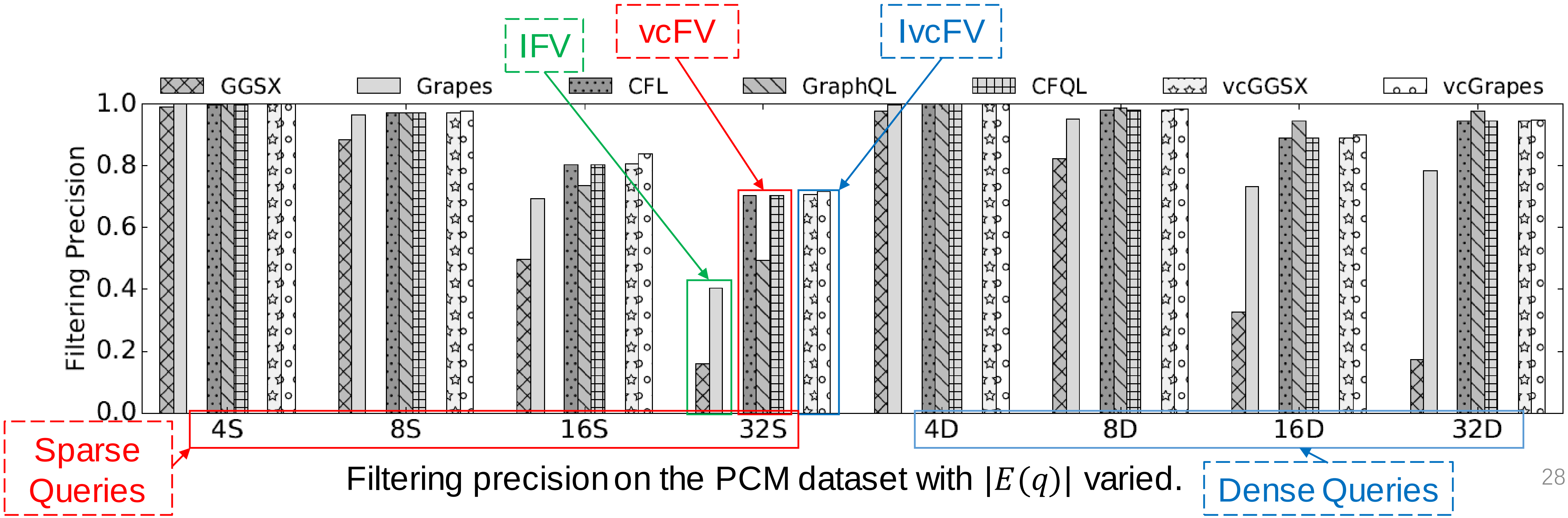
## Synthetic Datasets:

Set $|\Sigma| = 20$, $|d(G)| = 8$, $|V(G)| = 200$ and $|D| = 1000$ as default.

- Vary $|\Sigma|$ from 1, 10, 20, 40 to 80.
- Vary $d(G)$ from 4, 8, 16, 32 to 64.
- Vary $|V(G)|$ from 50, 200, 800, 3200 to 12800.
- Vary $|D|$ from $10^2$, $10^3$, $10^4$, $10^5$ to $10^6$.

# Filtering Precision

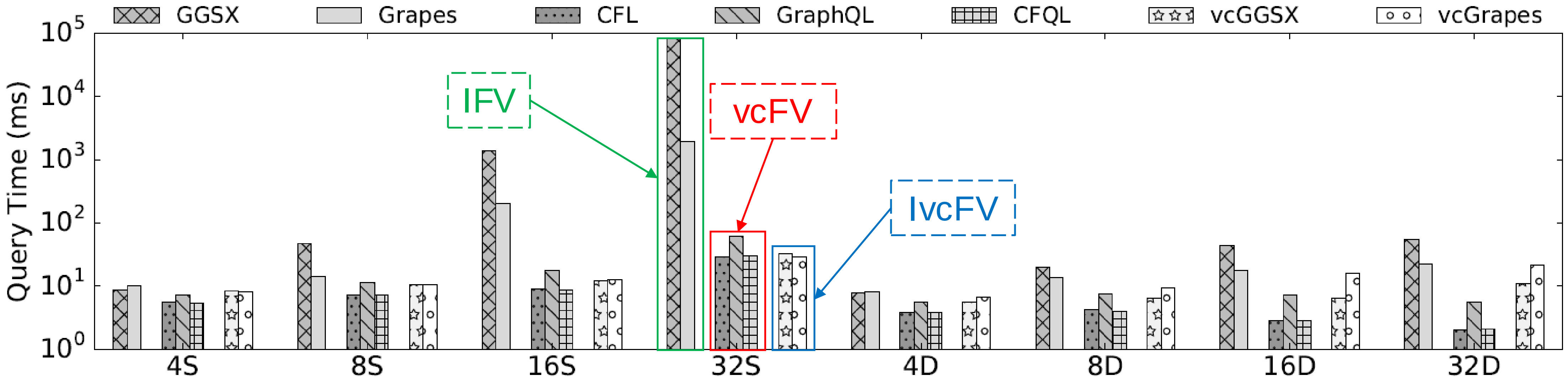
- $Filtering\ Precision = \frac{1}{|Q|}\sum_{q\in Q}\frac{|A(q)|}{|C(q)|}$ where $A(q)$ is the answer set and $C(q)$ is the candidate set.
- The filtering precision of vcFV algorithms is competitive with that of IFV algorithms.
- The filtering precision on dense query sets $(d(q) \geq 3)$ is higher than that on sparse query sets.




Filtering precision on the PCM dataset with $|E(q)|$ varied.

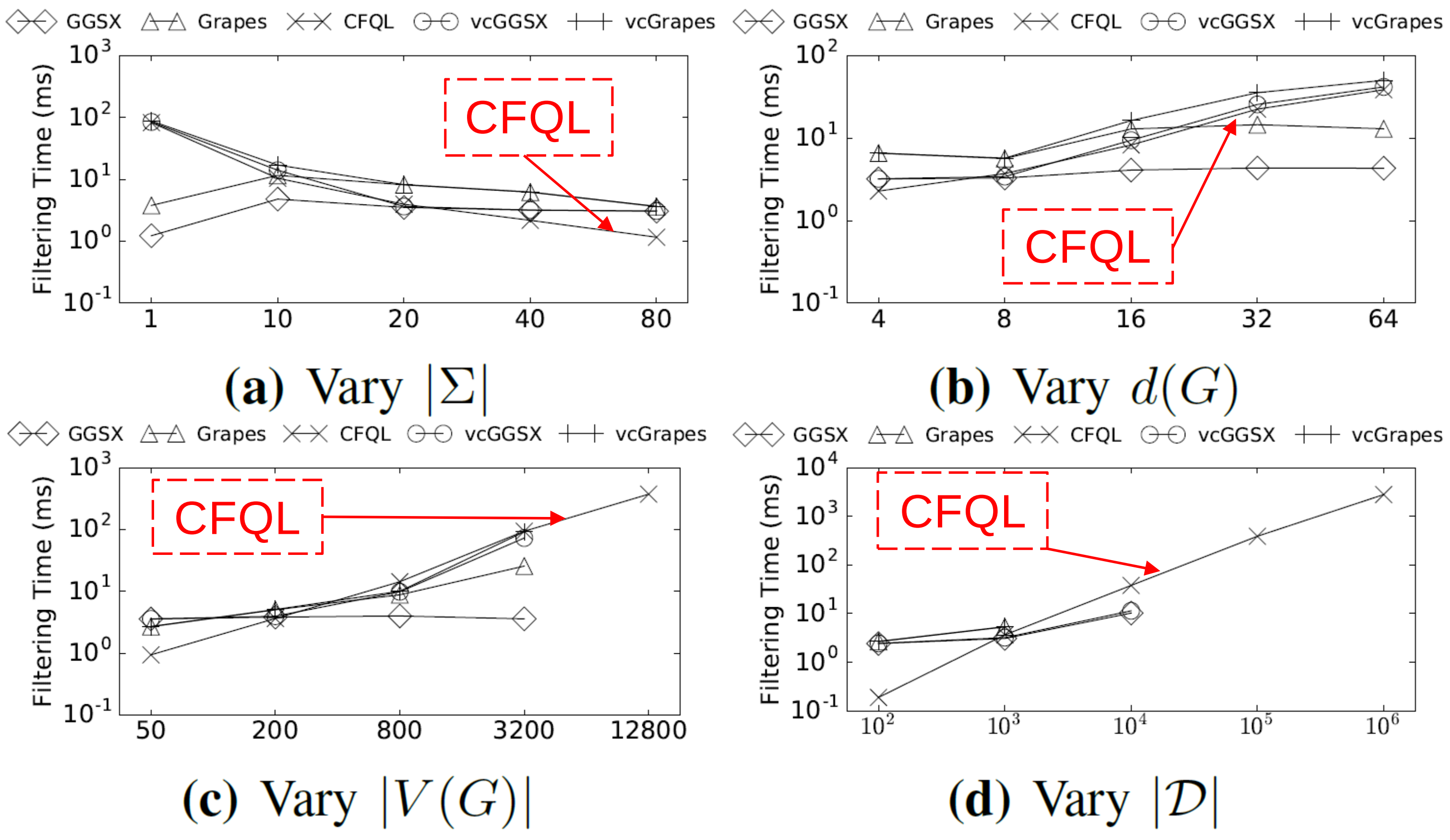
# Query Time

- Query time is the time spent on processing a query, consisting of both filtering and verification time.
- Benefiting from efficient subgraph matching, vcFV and IvcFV algorithms significantly outperform IFV algorithms.
- CFQL algorithm is competitive with vcGrapes and vcGGSX, the top-performing IFV algorithms integrated with the state-of-the-art subgraph matching algorithms.




Query time on PCM with $|E(q)|$ varying.

# Scalability




Filtering time on synthetic datasets with $Q_{8S}$.

- As the time complexity of the filtering method of CFQL is $O(|E(q)| \times |E(G)|)$, the filtering time of CFQL is roughly linear to $d(G)$, $|V(G)|$ and $|D|$.
- CFQL has a good scalability and completes the filtering on all test cases within 3 seconds.

# Conclusions

- The slow verification in existing IFV algorithms can lead us over-estimate the gain of filtering.
- vcFV algorithms are competitive in both filtering precision and time performance with that of top-performing IFV algorithms.
- vcFV algorithms can scale to hundreds of thousands of data graphs and graphs of thousands of vertices without any indices.

*Use vcFV algorithms instead of IFV algorithms!*

Scan me

*Download our source code and datasets from github by command:*

*git clone https://github.com/RapidsAtHKUST/SubgraphContainment.git*

# *Thanks. Q&A*

Scan me

*Download our source code and datasets from github by command:*

*git clone https://github.com/RapidsAtHKUST/SubgraphContainment.git*

# Why Separate the Process into Two Steps?

# Why Separate the Process into Two Steps?

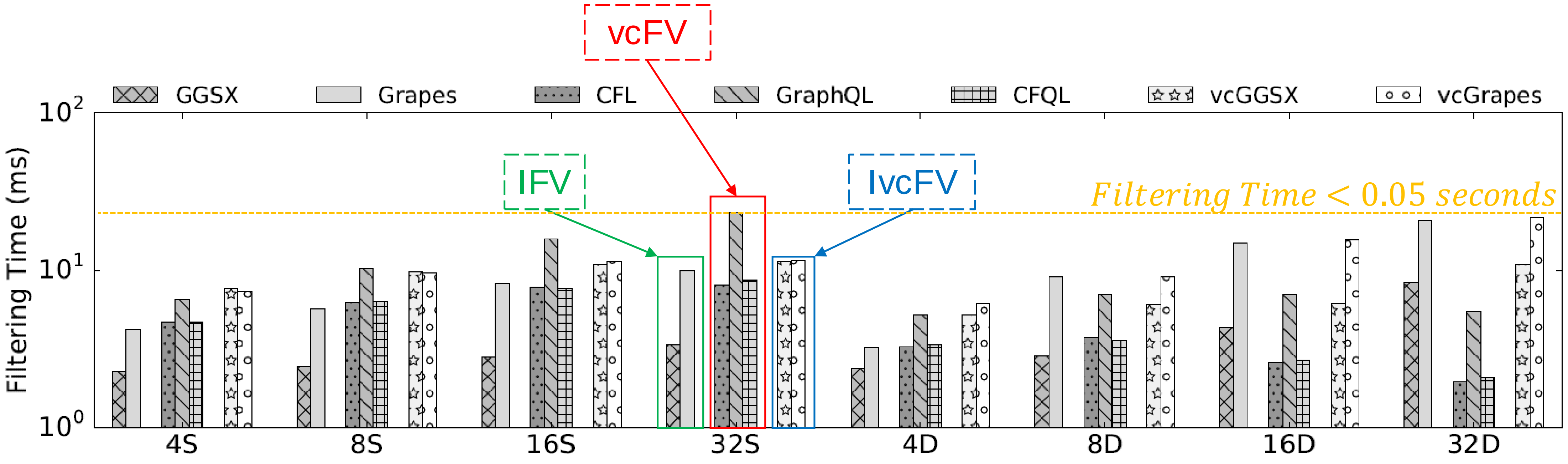
- ☺ Examine the bottleneck of subgraph query processing.

# Why Separate the Process into Two Steps?

- ☺ Examine the bottleneck of subgraph query processing.
- ☺ Combine different filtering and verification functions.

# Filtering Time

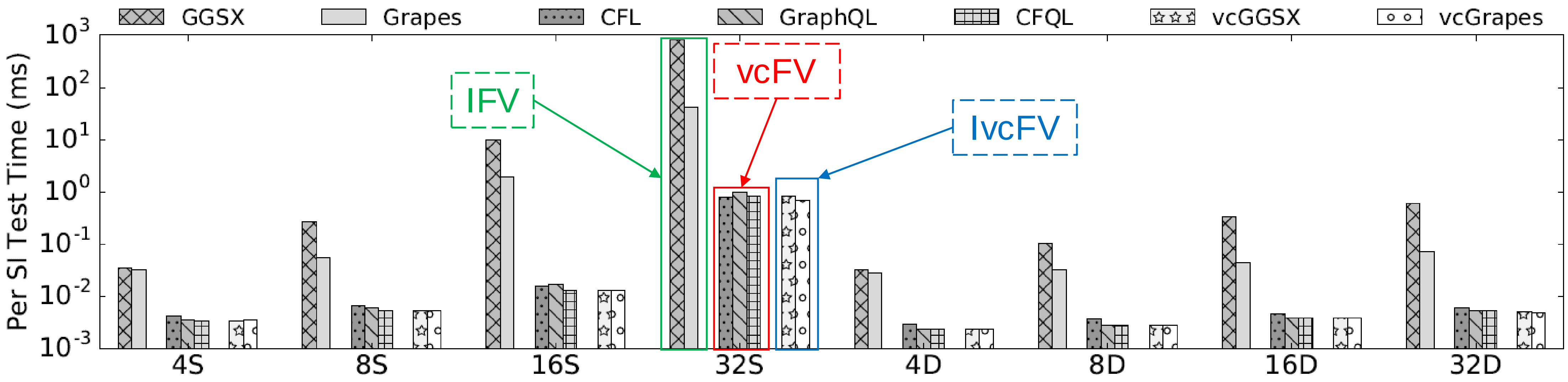
- Filtering time is the time spent on the filtering step.
- There is no single winner on all cases.
- The absolute value of the filtering time is very small.




Filtering time on PCM with $|E(q)|$ varying.

# Per SI Test Time

- $Per\ SI\ Test\ Time = \frac{1}{|Q|}\sum_{q \in Q}\frac{T_V(D,q)}{|C(q)|}$ where $T_v(D,q)$ is the time spent on the verification step.
- Efficient subgraph matching leads to the significant performance improvement of the verification.
- The slow verification in IFV algorithms can lead us to over-estimate the gain of filtering.
- Although the SI test is NP-complete, the filtering time can dominate the query time.




Per SI test time on PCM with $|E(q)|$ varying.

# Experimental Setup

## Algorithm Configuration:

- Grapes, vcGrapes: Use 6 threads and enumerate paths of up to a length of 4.
- GGSX, vcGGSX: Enumerate paths of up to a length of 4.
- CT-Index: Enumerate trees and cycles of up to a length of 4.

## Experiment Environment:

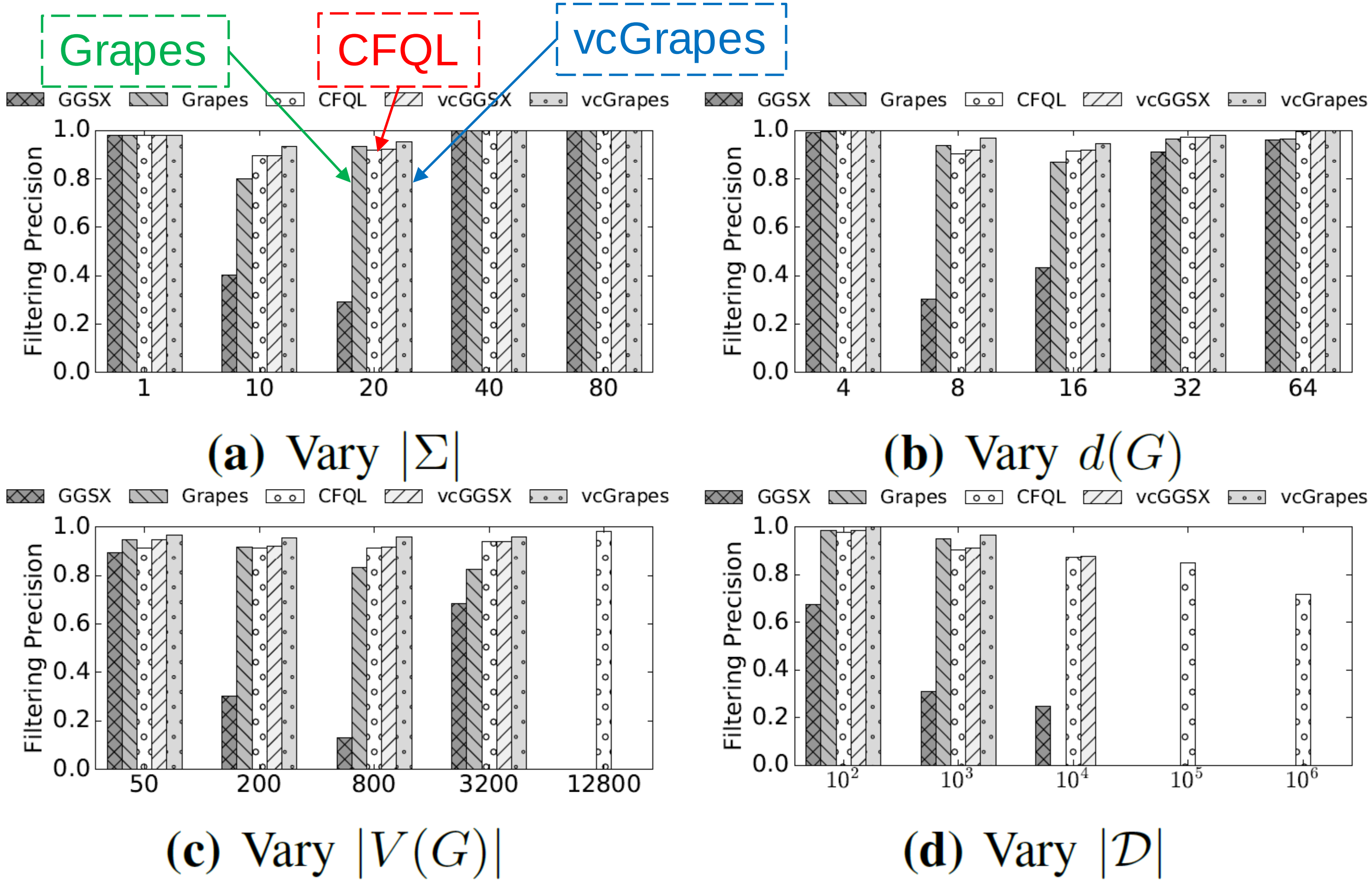
- The binary of CT-Index is implemented by JAVA, while the other algorithms are implemented in C++.
- Perform all experiments on a 64-bit Linux machine equipped with two Intel Xeon E5-2650 V3 processors and 64GB RAM.

# Experimental Setup

## Query Sets:

- Given a dataset, we generate 8 query sets including 4 dense query sets and 4 sparse query sets.
- Each query set contains 100 query graphs with the same number of edges.
- The number of edges varies from 4, 8, 16 to 32. We use $Q_{iD}$ and $Q_{iS}$ to denote dense query sets and sparse query sets with $i$ edges respectively.

# Scalability




Filtering precision on synthetic datasets with $Q_{8S}$.

- CFQL is competitive with the top-performing IFV and IvcFV algorithms on all cases.
- The filtering precision of CFQL is reasonable, which is greater than 0.75 on all cases.

# Backup

| | AIDS | | | | | | | | PDBS | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $Q_{4S}$ | $Q_{8S}$ | $Q_{16S}$ | $Q_{32S}$ | $Q_{4D}$ | $Q_{8D}$ | $Q_{16D}$ | $Q_{32D}$ | $Q_{4S}$ | $Q_{8S}$ | $Q_{16S}$ | $Q_{32S}$ | $Q_{4D}$ | $Q_{8D}$ | $Q_{16D}$ | $Q_{32D}$ |
| $\|V\|$ **per** $q$ | 5.00 | 8.92 | 16.29 | 31.30 | 4.99 | 6.00 | 9.66 | 18.16 | 4.95 | 8.80 | 16.41 | 32.13 | 4.93 | 5.99 | 9.45 | 18.90 |
| $\|\Sigma\|$ **per** $q$ | 2.41 | 2.88 | 3.58 | 4.15 | 2.37 | 2.11 | 1.99 | 2.76 | 2.43 | 2.96 | 3.45 | 3.73 | 2.45 | 2.00 | 2.56 | 3.23 |
| $d$ **per** $q$ | 1.60 | 1.80 | 1.97 | 2.05 | 1.60 | 2.67 | 3.32 | 3.62 | 1.62 | 1.83 | 1.96 | 2.00 | 1.63 | 2.67 | 3.41 | 3.44 |
| **% of trees** | 1.00 | 0.92 | 0.38 | 0.14 | 0.99 | 0.00 | 0.00 | 0.00 | 0.95 | 0.85 | 0.72 | 0.58 | 0.93 | 0.00 | 0.00 | 0.00 |

| | PCM | | | | | | | | PPI | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $Q_{4S}$ | $Q_{8S}$ | $Q_{16S}$ | $Q_{32S}$ | $Q_{4D}$ | $Q_{8D}$ | $Q_{16D}$ | $Q_{32D}$ | $Q_{4S}$ | $Q_{8S}$ | $Q_{16S}$ | $Q_{32S}$ | $Q_{4D}$ | $Q_{8D}$ | $Q_{16D}$ | $Q_{32D}$ |
| $\|V\|$ **per** $q$ | 4.96 | 8.48 | 15.49 | 29.47 | 4.00 | 5.20 | 7.20 | 10.11 | 4.98 | 8.75 | 16.13 | 30.86 | 4.51 | 5.60 | 9.02 | 16.44 |
| $\|\Sigma\|$ **per** $q$ | 4.34 | 6.64 | 10.18 | 14.06 | 3.65 | 4.43 | 5.82 | 7.77 | 3.28 | 4.71 | 6.63 | 8.43 | 3.20 | 3.86 | 4.85 | 6.57 |
| $d$ **per** $q$ | 1.62 | 1.92 | 2.09 | 2.19 | 2.00 | 3.09 | 4.46 | 6.39 | 1.61 | 1.83 | 1.99 | 2.09 | 1.80 | 2.88 | 3.59 | 4.06 |
| **% of trees** | 0.96 | 0.66 | 0.22 | 0.04 | 0.00 | 0.00 | 0.00 | 0.00 | 0.98 | 0.77 | 0.50 | 0.22 | 0.51 | 0.00 | 0.00 | 0.00 |

Statistics of query sets on the real-world datasets.

# Backup

| | AIDS | PDBS | PCM | PPI |
|---|---|---|---|---|
| **CT-Index** | 225 | 1,714 | OOT | OOT |
| **GGSX** | 8 | 5 | 433 | 2,209 |
| **Grapes** | 6 | 1 | 66 | 223 |

Indexing time on real-world datasets (seconds).

| | AIDS | PDBS | PCM | PPI |
|---|---|---|---|---|
| **Datasets** | 28.1 | 27.5 | 7.2 | 4.2 |
| **CFQL** | 0.055 | 3.627 | 0.150 | 2.576 |
| **CT-Index** | 338 | 317 | N/A | N/A |
| **GGSX** | 109 | 4 | 1,138 | 146 |
| **Grapes** | 254 | 72 | 3,302 | 831 |

Memory cost on real-world datasets (MB).

# Experiment Results on Indexing

| $\|\Sigma\|$ | **1** | **10** | **20** | **40** | **80** |
|---|---|---|---|---|---|
| **CT-Index** | OOT | OOT | OOT | OOT | OOT |
| **GGSX** | 28 | 105 | 194 | 184 | 167 |
| **Grapes** | 5 | 23 | 76 | 105 | 140 |
| $d(G)$ | **4** | **8** | **16** | **32** | **64** |
| **CT-Index** | 9,653 | OOT | OOT | OOT | OOT |
| **GGSX** | 26 | 216 | 1,131 | 4,220 | 18,541 |
| **Grapes** | 13 | 79 | 275 | 706 | 2,807 |
| $\|V(G)\|$ | **50** | **200** | **800** | **3200** | **12800** |
| **CT-Index** | OOT | OOT | OOT | OOT | OOT |
| **GGSX** | 35 | 198 | 1,078 | 1,830 | 8,630 |
| **Grapes** | 18 | 74 | 267 | 464 | OOM |
| $\|\mathcal{D}\|$ | $\mathbf{10^2}$ | $\mathbf{10^3}$ | $\mathbf{10^4}$ | $\mathbf{10^5}$ | $\mathbf{10^6}$ |
| **CT-Index** | 62,178 | OOT | OOT | OOT | OOT |
| **GGSX** | 11 | 174 | 3,256 | OOM | OOM |
| **Grapes** | 7 | 73 | OOM | OOM | OOM |

Table 1. The indexing time on synthetic datasets (seconds).

- Default Settings: $|\Sigma| = 20$, $|d(G)| = 8$, $|V(G)| = 200$ and $|D| = 1000$.
- OOT: out-of-time (24 Hours).
- OOM: out-of-memory (64GB).

# Backup

| Vary $|\Sigma|$ | **1** | **10** | **20** | **40** | **80** |
|---|---|---|---|---|---|
| **Datasets** | 7.8 | 7.8 | 7.8 | 7.8 | 7.8 |
| **CFQL** | 0.0320 | 0.0258 | 0.0237 | 0.0214 | 0.0348 |
| **GGSX** | 0.3242 | 649 | 3,650 | 5,937 | 10,502 |
| **Grapes** | 0.5586 | 1,171 | 6,443 | 10,267 | 15,483 |
| **Vary $d(G)$** | **4** | **8** | **16** | **32** | **64** |
| **Datasets** | 4.8 | 7.8 | 13.7 | 25.9 | 50.2 |
| **CFQL** | 0.0222 | 0.0214 | 0.0167 | 0.0160 | 0.0180 |
| **GGSX** | 779 | 3,668 | 8,532 | 9,842 | 9,957 |
| **Grapes** | 1,360 | 6,475 | 15,181 | 17,756 | 18,152 |
| **Vary $|V(G)|$** | **50** | **200** | **800** | **3200** | **12800** |
| **Datasets** | 1.9 | 7.7 | 35.8 | 122.1 | 491.6 |
| **CFQL** | 0.0044 | 0.0205 | 0.1679 | 0.1857 | 1.0286 |
| **GGSX** | 1,116 | 3,679 | 7,723 | 2,607 | 2,608 |
| **Grapes** | 1,940 | 6,492 | 19,265 | 19,658 | N/A |
| **Vary $|\mathcal{D}|$** | $\mathbf{10^2}$ | $\mathbf{10^3}$ | $\mathbf{10^4}$ | $\mathbf{10^5}$ | $\mathbf{10^6}$ |
| **Datasets** | 0.8 | 8 | 77 | 778 | 7,787 |
| **CFQL** | 0.0193 | 0.0227 | 0.0387 | 0.0424 | 0.0437 |
| **GGSX** | 381 | 3,647 | 36,317 | N/A | N/A |
| **Grapes** | 661 | 6,437 | N/A | N/A | N/A |

Memory cost on synthetic datasets (MB).

# Selected References


[1]. F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. In PVLDB, 2015.

[2]. K. Klein, N. Kriege, and P. Mutzel. Ct-index: Fingerprint-based graph indexing combining cycles and trees. In ICDE, 2011.

[3]. V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. In Pattern Recognition in Bioinformatics, 2010.

[4]. R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. In PloS one, 2013.

[5]. L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. In IEEE Transactions on Pattern Analysis and Machine Intelligence, 2004.

[6]. F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In SIGMOD, 2016.

[7]. H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In SIGMOD, 2008.