Appendix A.1: Time Complexity Analysis

(1) Neo4j-Based Subgraph Isomorphism Computation: Subgraph isomorphism in a property graph G with n nodes and m edges, given a query pattern Q with $|V_Q|$ nodes and $|E_Q|$ edges, is in general an NP-hard problem. However, when using Neo4j (via Cypher queries) for pattern matching, the complexity can be approximated by scanning the graph for each element of the pattern. In the worst case, the query engine may have to traverse $O(m)$ relationships for each of the $|E_Q|$ edges in Q, leading to a time complexity on the order of $O(m \cdot |E_Q|)$. In practice, $|E_Q|$ is typically much smaller than m (the patterns derived from Graph Differential Dependencies are small), so this cost approaches $O(m)$ – essentially linear in the size of the graph. (If the query pattern includes labels or indexed properties, Neo4j can leverage them to narrow down matches, but in the absence of highly selective indices, one can assume a linear scan per pattern edge in the worst case.) Thus, executing the subgraph isomorphism query in Neo4j will generally scale linearly with the graph size for small patterns, but it could grow linearly with $|E_Q|$ for more complex patterns.
(2) Impact of GDD Filtering and Blocking: FastER introduces Graph Differential Dependencies (GDDs) as constraints to drastically reduce the search space before performing expensive comparisons. Instead of searching the entire graph G, GDDs focus the computation on a query-relevant subgraph. Concretely, suppose applying the GDD rules yields a set of C candidate node pairs (potentially matching entity pairs), where $C \ll m$. This means only C pairs need to be considered further, as opposed to $\Theta(n^2)$ or other large combinations without filtering. Checking each candidate pair against the set of d GDD constraints incurs a cost of $O(C \cdot d)$ in total (each constraint check is $O(1)$ or a small constant factor, so this is effectively $O(C)$). After evaluating all GDD conditions, only a fraction of the pairs survive the filtering. Let C' denote the number of pairs remaining after GDD filtering; empirically we can express $C' = \beta\, C$ with a small fraction $\beta \ll 1$. These C' filtered candidates are then organized using a blocking strategy. FastER builds a blocking graph where nodes represent entity profiles and edges represent candidate matches between profiles. Constructing this graph and computing a similarity weight for each of the C' edges takes $O(C' \cdot r)$ time, where r is the number of attributes or features used to calculate the edge weight (this is linear in C' if r is a fixed small number of features). After weight computation, the candidate pairs (edges) are sorted by their similarity score, which adds an $O(C' \log C')$ overhead. The combination of GDD-based filtering and blocking ensures that the number of comparisons to be performed is vastly reduced and that those comparisons are structured: instead of considering all possible pairs, we only deal with C' high-likelihood pairs, and we have them sorted by likelihood for efficient processing. In summary, these steps reduce the problem size from m (or worse) down to C' and impose only linear or near-linear overheads in doing so.
(3) Progressive Profile Scheduling (PPS) Optimization: After filtering and blocking, FastER employs Progressive Profile Scheduling (PPS) to perform the actual entity matching in an efficient, incremental fashion. PPS processes

the sorted list of candidate pairs in descending order of similarity, which means it attempts the most likely matches first. This progressive approach has two major benefits for time complexity. First, it produces results early (useful for real-time "on-demand" requirements), and second, it enables threshold-based pruning of comparisons. In practice, we set a similarity threshold such that if a candidate pair's weight falls below this threshold, it is very unlikely to be an actual match and can be skipped entirely. Because the candidates are processed from highest to lowest similarity, once we reach pairs below the threshold, we can stop further comparisons in that block or for that profile. This significantly limits the number of comparisons each profile (entity) participates in. Rather than comparing each profile with all its C' potential matches, each profile will on average only be compared with a small constant number k' of top-ranked candidates (until the threshold condition causes pruning). If N is the number of profiles under consideration (e.g. the number of query-target entities that need to be resolved on demand), the comparison cost under PPS becomes $O(N \cdot k')$. Here k' can be viewed as the average number of comparisons per profile after early pruning (in many cases, k' is effectively bounded by a small value, since true duplicates for an entity tend to be limited in number or caught in the first few highest-similarity links). Moreover, as matches are found, transitive matching can further reduce comparisons: if profile A matches B, and B matches C, PPS can infer A matches C without a direct comparison, avoiding redundant checks. This transitivity (clustering duplicates as they are discovered) often reduces the effective k' even more. Overall, PPS ensures that the matching phase scales roughly linearly with the number of target entities, and the constant of proportionality k' is kept low by prioritization and pruning. This is a substantial improvement over a naive approach that might require comparing every candidate pair (which would be on the order of C' or worse per profile without PPS).

(4) Overall Time Complexity Estimation: Combining all the above components, we can derive the overall time complexity of the FastER framework. Summing the costs of subgraph pattern matching, GDD constraint filtering, blocking graph construction, and progressive matching, we get:

$$T(n) = O\big(m \cdot |E_Q|\big) \; + \; O(C \cdot d) \; + \; O(C' \log C') \; + \; O(N \cdot k') \; ,$$

where each term corresponds to the stages analyzed: subgraph isomorphism on the original graph ($O(m \cdot |E_Q|)$), GDD filtering of C candidates ($O(C \cdot d)$, which effectively becomes $O(C)$ and then $O(\beta C)$ after filtering), sorting and preparation of C' candidate pairs ($O(C' \log C')$), and the progressive comparisons ($O(N \cdot k')$). For typical use cases, $|E_Q|$ and d are small constants (small pattern and a fixed number of GDD rules), and β, k' are small fractions or constants reflecting the effectiveness of filtering and pruning. Thus, this complexity can be simplified to:

$$O\Big(m \; + \; \beta C \; + \; C' \log C' \; + \; N \cdot k'\Big) \; ,$$

noting that $C' = \beta C \ll C$. In an ideal scenario (best case), the filtering is highly selective (β is extremely small) and the similarity threshold is high enough

that k' remains very low. In such cases, the later terms become negligible, and the dominant cost is just scanning the relevant portion of the graph, yielding near-linear time $O(m)$ for the entire process. In a more conservative scenario (average case), we still expect β and k' to be significantly less than 1 (for instance, filtering might cut candidate pairs down to only a few percent, and each entity only needs a handful of comparisons), making the overall complexity close to linear in the size of the input graph and the number of query entities. Even in a worst-case scenario where the filtering is less effective ($\beta \approx 1$) and the threshold is low (large k'), the complexity $O(m + C' \log C' + N \cdot k')$ remains far more tractable than the naive $O(n^2)$ pairwise comparison of all entities. In summary, FastER's design leverages structural constraints (GDDs), intelligent blocking, and progressive processing to achieve a much lower time complexity than brute-force entity resolution, especially benefiting from on-demand focus (small N) and the typically sparse nature of true duplicate links (small k').

Appendix A.2: GDD Rules and Their Cypher Implementations

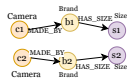
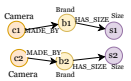
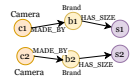
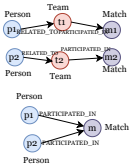
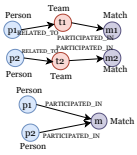
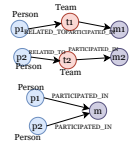
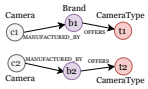
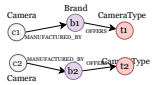
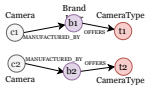
Dateset--rule	Graph pattern $Q[\bar{u}]$	constraints $\Phi_X \rightarrow \Phi_Y$	Cypher query statement	query recall
Altosight-rule1		φ: { δ_model(c1.model, c2.model) = 0 OR | intersection(split(c1.description), split(c2.description)) | > 0 } → δ_eid(c1, c2) = 0	MATCH (c1:Camera)-[:MADE_BY]->(b1:Brand), (c2:Camera)-[:MADE_BY]->(b2:Brand), (b1)-[:HAS_SIZE]->(s1:Size), (b2)-[:HAS_SIZE]->(s2:Size) WHERE (c1.model = c2.model OR (c1.description <> 'N\\A' AND c2.description <> 'N\\A' AND size(apoc.coll.intersection(apoc.text.split(c1.description, '\\s+'), apoc.text.split(c2.description, '\\s+'))) > 0)) RETURN c1, c2, s1, s2, b1, b2	1
Altosight-rule2		ϕ: { δ_model(c1.model, c2.model) = 0 } → δ_eid(c1, c2) = 0	MATCH (c1:Camera)-[:MADE_BY]->(b1:Brand), (c2:Camera)-[:MADE_BY]->(b2:Brand), (b1)-[:HAS_SIZE]->(s1:Size), (b2)-[:HAS_SIZE]->(s2:Size) WHERE (c1.model = c2.model) RETURN c1, c2, s1, s2, b1, b2	0.92
Altosight-rule3		ϕ: { | intersection(split(c1.description), split(c2.description)) | > 0 } → δ_eid(c1, c2) = 0	MATCH (c1:Camera)-[:MADE_BY]->(b1:Brand), (c2:Camera)-[:MADE_BY]->(b2:Brand), (b1)-[:HAS_SIZE]->(s1:Size), (b2)-[:HAS_SIZE]->(s2:Size) WHERE size(apoc.coll.intersection(apoc.text.split(c1.description, '\s+'), apoc.text.split(c2.description, '\s+'))) > 0)) RETURN c1, c2, s1, s2, b1, b2	0.96
WWC-rule1		ϕ1: { δ_levenshtein(p1.name, p2.name) > 0.75 } → δ_eid(p1, p2) = 0 ϕ2: { δ_levenshtein(p1.name, p2.name) > 0.7 } → δ_eid(p1, p2) = 0	MATCH (p1:Person)-[:RELATED_TO]->(t1:Team)-[:PARTICIPATED_IN]->(m1:Match), (p2:Person)-[:RELATED_TO]->(t2:Team)-[:PARTICIPATED_IN]->(m2:Match) WHERE apoc.text.levenshteinSimilarity(p1.name, p2.name) > 0.75 RETURN p1, p2, t1, t2, m1, m2 UNION MATCH (p1:Person)-[:PARTICIPATED_IN]->(m:Match), (p2:Person)-[:PARTICIPATED_IN]->(m:Match) WHERE p1 <> p2 AND apoc.text.levenshteinSimilarity(p1.name, p2.name) > 0.7 AND m.stage = 'Round of 16' RETURN p1, p2, null AS t1, null AS t2, m AS m1, m AS m2	0.967
WWC-rule2		ϕ1: { δ_levenshtein(p1.name, p2.name) > 0.75 } → δ_eid(p1, p2) = 0 ϕ2: { δ_levenshtein(p1.name, p2.name) > 0.9 } → δ_eid(p1, p2) = 0	MATCH (p1:Person)-[:RELATED_TO]->(t1:Team)-[:PARTICIPATED_IN]->(m1:Match), (p2:Person)-[:RELATED_TO]->(t2:Team)-[:PARTICIPATED_IN]->(m2:Match) WHERE apoc.text.levenshteinSimilarity(p1.name, p2.name) > 0.75 AND (m1.stage = 'Round of 16' AND m2.stage = 'Round of 16') RETURN p1, p2, t1, t2, m1, m2 UNION MATCH (p1:Person)-[:PARTICIPATED_IN]->(m:Match), (p2:Person)-[:PARTICIPATED_IN]->(m:Match) WHERE p1 <> p2 AND apoc.text.levenshteinSimilarity(p1.name, p2.name) > 0.9 AND m.stage = 'Round of 16' RETURN p1, p2, null AS t1, null AS t2, m AS m1, m AS m2	0.857
WWC-rule3		ϕ1: { c1.description ≠ 'N\A' ∧c2.description ≠ 'N\A' ∧ |intersection(split(c1.description), split(c2.description))| > 0 } → δ_eid(c1, c2) = 0 ϕ2: { c1.price = c2.price } → δ_eid(c1, c2) = 0	MATCH (c1:Camera)-[:MANUFACTURED_BY]->(b1:Brand), (c2:Camera)-[:MANUFACTURED_BY]->(b2:Brand),(b1)-[:OFFERS]->(t1:CameraType), (b2)-[:OFFERS]->(t2:CameraType) WHERE (b1.name CONTAINS 'leica' OR b2.name CONTAINS 'leica') AND ((c1.description <> 'N\A' AND c2.description <> 'N\A' AND size(apoc.coll.intersection(apoc.text.split(c1.description, '\s+'), apoc.text.split(c2.description, '\s+'))) > 0) OR c1.price = c2.price) RETURN c1, c2, t1, t2, b1, b2	0.923
SIGMOD20-rule1		ϕ1: { c1.description ≠ 'N\A' ∧ c2.description ≠ 'N\A' ∧ |intersection(split(c1.description), split(c2.description))| > 0 } → δ_eid(c1, c2) = 0 ϕ2: { c1.price = c2.price } → δ_eid(c1, c2) = 0	MATCH (c1:Camera)-[:MANUFACTURED_BY]->(b1:Brand), (c2:Camera)-[:MANUFACTURED_BY]->(b2:Brand), (b1)-[:OFFERS]->(t1:CameraType), (b2)-[:OFFERS]->(t2:CameraType) WHERE (b1.name CONTAINS 'leica' OR b2.name CONTAINS 'leica') AND ((c1.description <> 'N\A' AND c2.description <> 'N\A' AND size(apoc.coll.intersection(apoc.text.split(c1.description, '\s+'), apoc.text.split(c2.description, '\s+'))) > 0) OR c1.price = c2.price) RETURN c1, c2, t1, t2, b1, b2	1
SIGMOD20-rule2		ϕ1: { c1.description ≠ 'N\A' ∧ c2.description ≠ 'N\A' ∧ |intersection(split(c1.description), split(c2.description))| > 0 } → δ_eid(c1, c2) = 0 ϕ2: { c1.price = c2.price } → δ_eid(c1, c2) = 0	MATCH (c1:Camera)-[:MANUFACTURED_BY]->(b1:Brand), (c2:Camera)-[:MANUFACTURED_BY]->(b2:Brand), (b1)-[:OFFERS]->(t1:CameraType), (b2)-[:OFFERS]->(t2:CameraType) WHERE (b1.name CONTAINS 'leica' OR b2.name CONTAINS 'leica') AND ((c1.description <> 'N\\A' AND c2.description <> 'N\\A' AND size(apoc.coll.intersection(apoc.text.split(c1.description, '\\s+'), apoc.text.split(c2.description, '\\s+'))) > 0) OR c1.price = c2.price) RETURN c1, c2, t1, t2, b1, b2	0.805
SIGMOD20-rule3		ϕ1: { c1.description ≠ 'N\A' ∧ c2.description ≠ 'N\A' ∧ |intersection(split(c1.description), split(c2.description))| > 0 } → δ_eid(c1, c2) = 0 ϕ2: { c1.price = c2.price } → δ_eid(c1, c2) = 0	MATCH (c1:Camera)-[:MANUFACTURED_BY]->(b1:Brand), (c2:Camera)-[:MANUFACTURED_BY]->(b2:Brand), (b1)-[:OFFERS]->(t1:CameraType), (b2)-[:OFFERS]->(t2:CameraType) WHERE (b1.name CONTAINS 'leica' OR b2.name CONTAINS 'leica') AND ((c1.description <> 'N\\A' AND c2.description <> 'N\\A' AND size(apoc.coll.intersection(apoc.text.split(c1.description, '\\s+'), apoc.text.split(c2.description, '\\s+'))) > 0) OR c1.price = c2.price) RETURN c1, c2, t1, t2, b1, b2	0.916

Figure 1: GDD rule examples with Cypher implementation (1).

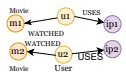
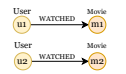
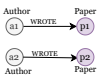
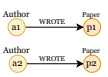
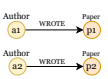
USES	Graph pattern $Q[\bar{u}]$	constraints $\Phi_X \rightarrow \Phi_Y$	Cypher query statement	query recall
Entity Resolution-rule1		ϕ_1: { u1.phone = u2.phone } → δ_eid(u1, u2) = 0 ϕ_2: { u1.last_name = u2.last_name } → δ_eid(u1, u2) = 0	MATCH (u1:User)-[:WATCHED]->(m1:Movie), (u2:User)-[:WATCHED]->(m2:Movie), (u1)-[:USES]->(ip1:IpAddress), (u2)-[:USES]->(ip2:IpAddress) WHERE u1 <> u2 AND u1.phone = u2.phone OR u1.last_name=u2.last_name RETURN u1, u2, m1, m2, ip1, ip2	0.667
Entity Resolution-rule2		ϕ: { u1.phone = u2.phone ∧ u1.last_name = u2.last_name } → δ_eid(u1, u2) = 0	MATCH (u1:User)-[:WATCHED]->(m1:Movie), (u2:User)-[:WATCHED]->(m2:Movie) WHERE u1 <> u2 AND u1.phone = u2.phone AND u1.last_name=u2.last_name RETURN u1, u2	1
Entity Resolution-rule3		ϕ: { u1.last_name = u2.last_name } → δ_eid(u1, u2) = 0	MATCH (u1:User), (u2:User) WHERE u1 <> u2 AND u1.last_name=u2.last_name RETURN u1, u2	1
DBLP-rule1		ϕ: { δ_levenshtein(a1.names, a2.names) > 0.4 } → δ_eid(a1, a2) = 0	MATCH (a1:Author)-[:WROTE]->(p1:Paper)-[:PUBLISHED_IN]->(v1:Venue), (a2:Author)-[:WROTE]->(p2:Paper)-[:PUBLISHED_IN]->(v2:Venue) WHERE v1.name CONTAINS 'acm' AND v2.name CONTAINS 'acm' AND apoc.text.levenshteinSimilarity(a1.names, a2.names)>0.4 RETURN a1, a2, p1, p2, v1, v2	0.682
DBLP-rule2		ϕ: {δ_levenshtein(p1.title, p2.title) > 0.6 } → δ_eid(a1, a2) = 0	MATCH (a1:Author)-[:WROTE]->(p1:Paper)-[:PUBLISHED_IN]->(v1:Venue), (a2:Author)-[:WROTE]->(p2:Paper)-[:PUBLISHED_IN]->(v2:Venue) WHERE v1.name CONTAINS 'acm' AND v2.name CONTAINS 'acm' AND apoc.text.levenshteinSimilarity(p1.title, p2.title) > 0.6 RETURN a1, a2, p1, p2, v1, v2	1
DBLP-rule3		ϕ: { δ_levenshtein(p1.title, p2.title) > 0.6 } → δ_eid(a1, a2) = 0	MATCH (a1:Author)-[:WROTE]->(p1:Paper)-[:PUBLISHED_IN]->(v1:Venue), (a2:Author)-[:WROTE]->(p2:Paper)-[:PUBLISHED_IN]->(v2:Venue) WHERE v1.name CONTAINS 'acm' AND v2.name CONTAINS 'acm' AND apoc.text.levenshteinSimilarity(p1.title, p2.title) > 0.9 RETURN a1, a2, p1, p2, v1, v2	0.992
Arxiv-rule1		ϕ: {δ_levenshteinDist(a1.name, a2.name) ≤ 2 } → δ_eid(a1, a2) = 0	MATCH (a1:Author)-[:WROTE]->(p1:Paper), (a2:Author)-[:WROTE]->(p2:Paper) WHERE a1 <> a2 AND a1.name contains 'robbert' AND a2.name contains 'robbert' AND apoc.text.levenshteinDistance(a1.name, a2.name) <=2 RETURN a1, a2, p1, p2	1
Arxiv-rule2		ϕ: { δ_levenshteinDist(a1.name, a2.name) ≤ 1 } → δ_eid(a1, a2) = 0	MATCH (a1:Author)-[:WROTE]->(p1:Paper), (a2:Author)-[:WROTE]->(p2:Paper) WHERE a1 <> a2 AND a1.name contains 'robbert' AND a2.name contains 'robbert' AND apoc.text.levenshteinDistance(a1.name, a2.name) <=1 RETURN a1, a2, p1, p2	1
Citeseer-rule1		ϕ: { a1.name = a2.name } → δ_eid(a1, a2) = 0	MATCH (a1:Author)-[:WROTE]->(p1:Paper), (a2:Author)-[:WROTE]->(p2:Paper) WHERE a1 <> a2 AND p1.title contains 'nowledge' and p2.title contains 'nowledge' AND a1.name=a2.name RETURN a1, a2, p1, p2	0.801
Citeseer-rule2		ϕ: { δ_levenshteinDist(a1.name, a2.name) ≤ 5 } → δ_eid(a1, a2) = 0	MATCH (a1:Author)-[:WROTE]->(p1:Paper), (a2:Author)-[:WROTE]->(p2:Paper) WHERE a1 <> a2 AND p1.title contains 'nowledge' and p2.title contains 'nowledge' AND apoc.text.levenshteinDistance(a1.name, a2.name) <=5 RETURN a1, a2, p1, p2	1
Citeseer-rule3		ϕ: { δ_levenshteinDist(a1.name, a2.name) ≤ 4 } → δ_eid(a1, a2) = 0	MATCH (a1:Author)-[:WROTE]->(p1:Paper), (a2:Author)-[:WROTE]->(p2:Paper) WHERE a1 <> a2 AND p1.title contains 'nowledge' and p2.title contains 'nowledge' AND apoc.text.levenshteinDistance(a1.name, a2.name) <=4 RETURN a1, a2, p1, p2	0.972

Figure 2: GDD rule examples with Cypher implementation (2).

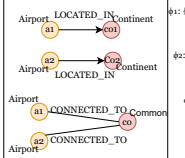
Dateset--rule	Graph pattern $Q[\bar{u}]$	constraints $\Phi_X \rightarrow \Phi_Y$	Cypher query statement	query recall
Amazon-rule1		ϕ: { (|intersection(split(s1.name, " "))| > 1 ∨ δ_levenshteinDist(s1.name, s2.name) < 6) ∧ |toFloat(s1.price) - toFloat(s2.price)| < 3 } → δ_eid(s1, s2) = 0	MATCH (s1:Software)-[:MADE_BY]->(m1:Manufacturer), (s2:Software)-[:MADE_BY]->(m2:Manufacturer) WHERE (toFloat(s1.price)<10 AND toFloat(s2.price)<10) AND ((size(apoc.coll.intersection(apoc.text.split(s1.name, ' '), apoc.text.split(s2.name, ' '))) >1 OR apoc.text.levenshteinDistance(s1.name, s2.name) < 6) AND abs(toFloat(s1.price) - toFloat(s2.price)) < 3) RETURN s1, s2, m1, m2	0.855
Fodors-Zagats-rule1		ϕ: { "new york" ∈ c1.value ∧ "new york" ∈ c2.value ∧ SIZE(intersection(split(r1.name, " "), split(r2.name, " "))) > 0 } → δ_eid(r1, r2) = 0	MATCH (r1:Restaurant)-[:LOCATED_AT]->(a1:Address)-[:IN_CITY]->(c1:City), (r2:Restaurant)-[:LOCATED_AT]->(a2:Address)-[:IN_CITY]->(c2:City) WHERE c1.value CONTAINS 'new york' AND c2.value CONTAINS 'new york' AND r1.id < r2.id WITH r1, r2, apoc.text.split(r1.name, ' ') AS r1Words, apoc.text.split(r2.name, ' ') AS r2Words, a1, a2, c1, c2 WHERE SIZE(apoc.coll.intersection(r1Words, r2Words)) > 0 RETURN r1, r2, a1, a2, c1, c2	0.970
GDS-rule3		ϕ_1: { δ_levenshteinDist(a1.city, a2.city) < 2 } → δ_eid(a1, a2) = 0 ϕ_2: {SIZE(intersection(split(a1.name, " "), split(a2.name, " "))) > 4 } → δ_eid(a1, a2) = 0 ϕ_3: { δ_levenshteinDist(a1.name, a2.name) < 3 } → δ_eid(a1, a2) = 0 ϕ_4: { a1.number = a2.number } → δ_eid(a1, a2) = 0	MATCH (a1:Airport)-[:LOCATED_IN]->(co1:Continent), (a2:Airport)-[:LOCATED_IN]->(co2:Continent) WHERE co1.name CONTAINS 'EU' AND co2.name CONTAINS 'EU' AND a1 <> a2 // Ensure distinct airports AND ((a1.city IS NOT NULL AND a2.city IS NOT NULL AND apoc.text.levenshteinDistance(a1.city, a2.city) < 2) OR (size(apoc.coll.intersection(apoc.text.split(a1.name, ' '), apoc.text.split(a2.name, ' '))) >4 OR apoc.text.levenshteinDistance(a1.name, a2.name) < 3) OR (a1.number IS NOT NULL AND a2.number IS NOT NULL AND a1.number = a2.number)) OPTIONAL MATCH (a1)-[:CONNECTED_TO]-(common:Airport)-[:CONNECTED_TO]-(a2) RETURN a1, a2, co1, co2	0.99
GDS-rule1		ϕ_1: { c1 = c2 } → δ_eid(a1, a2) = 0 ϕ_2: { c1 = c2 ∧ δ_jw(a1.city, a2.city) < 0.55 } → δ_eid(a1, a2) = 0 ϕ_3: { c1 = c2 ∧ δ_jw(a1.name, a2.name) < 0.46 }	MATCH (a1:Airport)-[:LOCATED_IN]->(c1:City), (a2:Airport)-[:LOCATED_IN]->(c2:City) WHERE c1 = c2 AND (toInteger(a1.number) < 1000 OR toInteger(a2.number) < 1000) AND apoc.text.jaroWinklerDistance(a1.city, a2.city) < 0.55 AND apoc.text.jaroWinklerDistance(a1.name, a2.name) < 0.46 RETURN a1, a2, c1, c2	0.791

Figure 3: GDD rule examples with Cypher implementation (3).